Parameters and Arguments

Introduction

A function becomes much more useful when we can give it information.

For example, imagine a function that says hello:

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

This always prints the same message.

But what if we want it to greet different people?

```
Hello Hafsa
Hello Asim
Hello Ali
```

Instead of creating three different functions, we can give information to the same function.

This is where **parameters and arguments** come in.

```
Function
   ↓
Receives information
   ↓
Processes the information
   ↓
Produces a result
```

---

1. What Is a Parameter?

A **parameter** is a variable listed inside a function's parentheses when the function is defined.

Example:

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

Here:

```
name
```

is a **parameter**.

Think of a parameter as a placeholder for information that the function will receive later.

```
Parameter = placeholder
```

The function does not know the actual name yet.

It simply knows:

> "When someone calls me, they need to give me a name."

---

2. What Is an Argument?

An **argument** is the actual value that we pass to a function when calling it.

Example:

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


greet("Hafsa")
```

Here:

```
name    → parameter
"Hafsa" → argument
```

So:

```
Parameter = variable in the function definition
Argument  = actual value passed during the function call
```

---

3. Parameter vs Argument

Compare these two lines:

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

and:

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

The first one defines the parameter:

```
name
```

The second one provides the argument:

```
"Hafsa"
```

A simple way to remember:

```
Parameter → What the function expects
Argument  → What you actually give it
```

---

4. A Simple Example

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


greet("Hafsa")
```

Output:

```
Hello Hafsa
```

When Python runs:

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

the value `"Hafsa"` is assigned to the parameter `name`.

Conceptually:

```
name = "Hafsa"
```

Then Python executes:

```
print("Hello", name)
```

Result:

```
Hello Hafsa
```

---

5. Why Do We Need Parameters?

Parameters make functions reusable.

Without parameters:

```
def greet_hafsa():
    print("Hello Hafsa")


def greet_asim():
    print("Hello Asim")
```

We need separate functions.

With a parameter:

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

Now one function can handle many people:

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

Output:

```
Hello Hafsa
Hello Asim
Hello Ali
```

The function is written once but can work with different data.

---

6. One Parameter

A function can have one parameter.

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

Call it:

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

Output:

```
25
```

Here:

```
number → parameter
5      → argument
```

Another call:

```
print(square(10))
```

Output:

```
100
```

The parameter allows the function to work with different values.

---

7. Multiple Parameters

A function can have multiple parameters.

Example:

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

Call:

```
result = add(10, 20)

print(result)
```

Output:

```
30
```

Here:

```
a → parameter
b → parameter

10 → argument
20 → argument
```

Conceptually:

```
a = 10
b = 20
```

Then:

```
return a + b
```

becomes:

```
return 10 + 20
```

Result:

```
30
```

---

8. Arguments Are Matched by Position

When using positional arguments, Python matches arguments with parameters according to their order.

Example:

```
def introduce(name, age):
    print("Name:", name)
    print("Age:", age)
```

Call:

```
introduce("Hafsa", 25)
```

Python matches them like this:

```
name → "Hafsa"
age  → 25
```

Because `"Hafsa"` is the first argument and `25` is the second argument.

---

9. Order Matters

Consider:

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

This:

```
subtract(10, 3)
```

produces:

```
7
```

But:

```
subtract(3, 10)
```

produces:

```
-7
```

The values are the same, but their positions are different.

Therefore:

```
Positional arguments → order matters
```

---

10. Different Data Types Can Be Arguments

Arguments don't have to be numbers.

They can be strings:

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

Numbers:

```
square(5)
```

Lists:

```
def show_items(items):
    print(items)


show_items(["Python", "JavaScript", "Django"])
```

Booleans:

```
def check_status(is_active):
    print(is_active)


check_status(True)
```

Python functions can receive different types of data depending on what the function is designed to do.

---

11. Arguments Can Be Variables

An argument does not have to be written directly inside the function call.

We can use a variable.

```
name = "Hafsa"

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


greet(name)
```

Here the variable contains the value:

```
name = "Hafsa"
```

and that value is passed to the function.

---

12. Expressions Can Also Be Arguments

We can pass an expression as an argument.

Example:

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


print(square(5 + 1))
```

Python first evaluates:

```
5 + 1
```

which gives:

```
6
```

Then the function receives:

```
6
```

So the result is:

```
36
```

---

13. Keyword Arguments

Python also allows us to specify which parameter should receive a value.

Example:

```
def introduce(name, age):
    print("Name:", name)
    print("Age:", age)


introduce(name="Hafsa", age=25)
```

Output:

```
Name: Hafsa
Age: 25
```

These are called **keyword arguments**.

Instead of relying only on position, we explicitly name the parameter.

---

14. Positional vs Keyword Arguments

Positional

```
introduce("Hafsa", 25)
```

Python uses position:

```
first  → name
second → age
```

Keyword

```
introduce(name="Hafsa", age=25)
```

Python uses the parameter names:

```
name → "Hafsa"
age  → 25
```

---

15. Keyword Arguments Can Improve Readability

Compare:

```
create_user("Hafsa", 25, "Pakistan")
```

with:

```
create_user(
    name="Hafsa",
    age=25,
    country="Pakistan"
)
```

The second version makes it easier to understand what each value represents.

This becomes especially useful when a function has many parameters.

---

16. Changing the Order With Keyword Arguments

With positional arguments:

```
def introduce(name, age):
    print(name, age)
```

This follows the parameter order:

```
introduce("Hafsa", 25)
```

With keyword arguments, we can change the order:

```
introduce(age=25, name="Hafsa")
```

Output:

```
Hafsa 25
```

Python knows which value belongs to which parameter because we specified the parameter names.

---

17. Default Parameters

Sometimes we want a parameter to have a default value.

Example:

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

If we call:

```
greet()
```

Output:

```
Hello Guest
```

If we provide an argument:

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

Output:

```
Hello Hafsa
```

The provided argument replaces the default value.

---

18. Why Are Default Parameters Useful?

Default parameters allow a function to work even when a particular argument isn't provided.

Example:

```
def create_profile(name, country="Pakistan"):
    print(name)
    print(country)
```

Call:

```
create_profile("Hafsa")
```

Output:

```
Hafsa
Pakistan
```

Or provide another country:

```
create_profile("Hafsa", "Canada")
```

Output:

```
Hafsa
Canada
```

---

19. Required vs Optional Arguments

Consider:

```
def create_profile(name, country="Pakistan"):
    print(name)
    print(country)
```

Here:

```
name
```

is a required parameter.

```
country
```

has a default value, so it is optional when calling the function.

This means:

```
create_profile("Hafsa")
```

works.

And:

```
create_profile("Hafsa", "Canada")
```

also works.

---

20. Default Parameters Must Come After Required Parameters

This is correct:

```
def create_profile(name, country="Pakistan"):
    print(name, country)
```

This is incorrect:

```
def create_profile(country="Pakistan", name):
    print(name, country)
```

In a normal function definition, parameters without defaults should come before parameters with defaults.

Think:

```
Required parameters
        ↓
Default parameters
```

---

21. Multiple Arguments

A function can receive several arguments.

Example:

```
def student_result(name, math, english, computer):
    print("Student:", name)
    print("Math:", math)
    print("English:", english)
    print("Computer:", computer)
```

Call:

```
student_result("Hafsa", 90, 85, 95)
```

Output:

```
Student: Hafsa
Math: 90
```

```
English: 85
Computer: 95
```

Each argument is assigned to the matching parameter.

---

22. What Happens If We Give Too Few Arguments?

Suppose:

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

Now:

```
add(10)
```

Python raises an error because the function requires two arguments.

Conceptually:

```
a → 10
b → missing
```

The function cannot perform:

```
a + b
```

because `b` has no value.

---

23. What Happens If We Give Too Many Arguments?

The opposite problem can also happen.

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

Calling:

```
add(10, 20, 30)
```

gives an error because the function has only two parameters but received three arguments.

```
Parameters → 2
Arguments  → 3
```

They don't match.

Later, Python provides special techniques such as `*args` for accepting a variable number of positional arguments.

---

24. Arguments and Return Values Work Together

Parameters and arguments provide information **to** a function.

`return` sends information **back**.

Example:

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

Call:

```
result = multiply(5, 4)
```

Flow:

```
5 ──────┐
        ↓
      multiply()
        ↓
      5 × 4
        ↓
       20
        ↓
     result
```

So we can think of a function as:

```
Arguments
    ↓
Function
    ↓
Return value
```

---

25. Practical Example: Shopping Cart

Imagine a shopping application.

We want to calculate the total price.

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

Call:

```
total = calculate_total(500, 3)

print(total)
```

Output:

```
1500
```

Here:

```
price    → parameter
quantity → parameter

500      → argument
3        → argument

1500     → returned value
```

---

26. Practical Example: Discount Calculator

```
def calculate_discount(price, discount):
    return price - (price * discount / 100)
```

Call:

```
final_price = calculate_discount(1000, 10)

print(final_price)
```

Output:

```
900.0
```

The function receives:

```
price = 1000
discount = 10
```

Then calculates the result.

---

27. Practical Example: haas.dev Resource

Imagine we want a function that creates basic information for a resource.

```
def show_resource(title, category):
    print("Title:", title)
    print("Category:", category)
```

Call:

```
show_resource(
    "Python Functions",
    "Python"
)
```

Output:

```
Title: Python Functions
Category: Python
```

We can reuse it:

```
show_resource("Python Lists", "Python")
show_resource("JavaScript Arrays", "JavaScript")
```

The function stays the same while the data changes.

---

28. Choosing Good Parameters

Parameter names should describe the information they represent.

Good:

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

Less clear:

```
def calculate_total(x, y):
```

Both technically work, but:

```
price
quantity
```

tell us what the values mean.

Use meaningful names when the purpose is not obvious.

---

29. Avoid Unnecessary Parameters

Don't add parameters just because you can.

Instead of:

```
def greet(name, age, country, color, phone, height):
    print("Hello", name)
```

If the function only needs the person's name, use:

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

A function should receive the information it actually needs.

---

30. Common Mistakes

Mistake 1: Confusing Parameters and Arguments

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

`name` is the parameter.

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

`"Hafsa"` is the argument.

Mistake 2: Wrong Number of Arguments

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


add(10)
```

One argument is missing.

Mistake 3: Passing Too Many Arguments

```
add(10, 20, 30)
```

The function only expects two arguments.

Mistake 4: Wrong Positional Order

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

```
divide(2, 10)
```

This produces:

```
0.2
```

It is different from:

```
divide(10, 2)
```

which produces:

```
5.0
```

Mistake 5: Using an Incorrect Keyword Name

Suppose:

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

This is correct:

```
greet(name="Hafsa")
```

But:

```
greet(username="Hafsa")
```

is incorrect because the function has no parameter called `username`.

---

31. Problem Solving Framework

When designing a function, ask:

Step 1: What does the function need?

```
Input
```

Step 2: Give each input a meaningful parameter name.

```
price
quantity
```

Step 3: Decide what the function should do.

```
price × quantity
```

Step 4: Decide whether it needs to return something.

```
total
```

Step 5: Create the function.

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

Step 6: Call it with actual arguments.

```
calculate_total(500, 3)
```

---

32. Mini Project: Student Grade Calculator

Create a function that receives a student's name and three marks.

```
def calculate_result(name, math, english, computer):
    total = math + english + computer
    average = total / 3

    print("Student:", name)
    print("Total:", total)
    print("Average:", average)
```

Call:

```
calculate_result("Hafsa", 90, 85, 95)
```

Output:

```
Student: Hafsa
Total: 270
Average: 90.0
```

Now try:

```
calculate_result("Asim", 75, 80, 85)
```

The same function works for another student.

---

33. 5 Minute Challenge

Create a function:

```
calculate_bill(price, quantity, discount)
```

It should:

1. Receive the product price.
2. Receive the quantity.
3. Receive the discount percentage.
4. Calculate the original total.
5. Apply the discount.
6. Return the final price.

Example:

```
total = calculate_bill(1000, 2, 10)

print(total)
```

Expected result:

```
1800
```

Think about the function before writing it:

```
Inputs:
price
quantity
discount

Processing:
price × quantity
then apply discount

Output:
final price
```

---

34. Practice Exercises

Exercise 1

Create:

```
greet(name)
```

It should print:

```
Hello, <name>
```

Exercise 2

Create:

```
add(a, b)
```

Return the sum.

Exercise 3

Create:

```
calculate_area(length, width)
```

Return the area of a rectangle.

Exercise 4

Create:

```
calculate_average(mark1, mark2, mark3)
```

Return the average.

Exercise 5

Create:

```
create_profile(name, age, country)
```

Display all three values.

Exercise 6

Create:

```
welcome(name="Guest")
```

If no argument is provided, it should print:

```
Welcome Guest
```

If the user calls:

```
welcome("Hafsa")
```

it should print:

```
Welcome Hafsa
```

---

35. Mini Quiz

Question 1

In this code:

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

What is `name`?

A. Argument

B. Parameter

C. Return value

D. Function call

Question 2

In this code:

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

What is `"Hafsa"`?

A. Parameter

B. Function

C. Argument

D. Keyword

Question 3

What happens here?

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

add(10)
```

A. Returns 10

B. Returns 0

C. Causes an error because an argument is missing

D. Prints 10

Question 4

Which is a keyword argument?

A.

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

B.

```
greet(name="Hafsa")
```

C.

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

D.

```
return name
```

Answers

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

---

36. Interview Questions

Beginner

1. What is a parameter?
2. What is an argument?
3. What is the difference between a parameter and an argument?
4. Why do functions use parameters?
5. Can a function have multiple parameters?
6. What happens if a required argument is missing?
7. What happens if too many arguments are provided?

Intermediate

8. What are positional arguments?
9. What are keyword arguments?
10. What is the difference between positional and keyword arguments?
11. What is a default parameter?
12. Why are default parameters useful?
13. Can keyword arguments change the order of values?
14. Why should function parameters have meaningful names?
15. How do arguments and return values work together?

---

37. Cheat Sheet

Parameter

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

`name` = parameter

Positional Argument

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

`"Hafsa"` = argument

Multiple Parameters

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

Multiple Arguments

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

Keyword Arguments

```
add(a=10, b=20)
```

Default Parameter

```
def greet(name="Guest"):
    print(name)
```

Using the Default

```
greet()
```

Replacing the Default

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

---

38. The Mental Model

Think of a function like a machine.

```
          ARGUMENTS
        ↓       ↓
```

```
      500       3
        ↓       ↓
    ┌─────────────────┐
    │ calculate_      │
    │ total()         │
    └─────────────────┘
            ↓
          1500
```

Inside the function:

```
500 → price
3   → quantity
```

So:

```
Arguments
   ↓
Parameters
   ↓
Function logic
   ↓
Return value
```

This is one of the most important patterns in programming.

---

39. Key Takeaways

- A **parameter** is a variable defined inside a function's parentheses.
- An **argument** is the actual value passed when calling the function.
- A function can have one or multiple parameters.
- Arguments can be values, variables, or expressions.
- Positional arguments are matched according to order.
- Keyword arguments explicitly identify the parameter receiving the value.
- Default parameters provide fallback values.
- Required parameters normally come before default parameters.
- The number of required arguments must match the function's expectations.
- Meaningful parameter names make functions easier to understand.
- Parameters provide information **to** a function.
- A `return` statement can send information **back** from the function.

The core idea:

```
PARAMETER
"What kind of information do I need?"

        ↓

ARGUMENT
"Here is the actual information."

        ↓

FUNCTION
"Now I can perform the task."
```

Visit haas.dev for more resources and guides.

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

Next Step

The next useful topic is **Return Values**, where we will focus specifically on how functions send results back to the code that called them.