Python Keyword Arguments

Introduction

When we call a Python function, we usually provide arguments based on their **position**.

For example:

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

introduce("Hafsa", 25)
```

Python understands:

```
"Hafsa" → name
25 → age
```

But Python also allows us to provide an argument using the **parameter's name**.

This is called a **keyword argument**.

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

Here:

- `name="Hafsa"` is a keyword argument.
- `age=25` is a keyword argument.

The main idea is:

> A keyword argument passes a value to a function by explicitly naming the parameter.

---

1. Positional Arguments vs Keyword Arguments

Consider this function:

```
def student_info(name, age, grade):
    print(name)
    print(age)
    print(grade)
```

Positional arguments

```
student_info("Hafsa", 25, 9)
```

Python matches values by position:

```
"Hafsa" → name
25 → age
9 → grade
```

Keyword arguments

```
student_info(
    name="Hafsa",
    age=25,
    grade=9
)
```

Now Python matches values by parameter name.

---

2. Why Are Keyword Arguments Useful?

Imagine a function with several parameters:

```
def create_profile(name, age, city, role):
    pass
```

Using positional arguments:

```
create_profile("Hafsa", 25, "Gujranwala", "Developer")
```

You need to remember the exact order.

With keyword arguments:

```
create_profile(
    name="Hafsa",
    age=25,
    city="Gujranwala",
    role="Developer"
)
```

The meaning of every value is immediately clear.

This makes code easier to read.

---

3. Keyword Arguments Make Code More Readable

Compare:

```
create_profile("Hafsa", 25, "Gujranwala", "Developer")
```

with:

```
create_profile(
    name="Hafsa",
    age=25,
    city="Gujranwala",
    role="Developer"
)
```

The second version clearly communicates what each value represents.

This becomes especially useful when a function has many parameters.

---

4. Changing the Order

One of the biggest benefits of keyword arguments is that you can change the order.

Consider:

```
def student_info(name, age, grade):
    print(name, age, grade)
```

With positional arguments:

```
student_info("Hafsa", 25, 9)
```

Changing the order would change the meaning.

But keyword arguments can be written in any order:

```
student_info(
    grade=9,
    name="Hafsa",
    age=25
)
```

Python still knows:

```
grade → 9
name → "Hafsa"
age → 25
```

because the parameter names are explicitly provided.

---

5. Keyword Arguments With Default Arguments

Keyword arguments are particularly useful with default parameters.

Example:

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

We can use the defaults:

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

Output:

```
Hafsa
Developer
Pakistan
```

But we can change only one default:

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

Output:

```
Hafsa
Developer
Canada
```

Or change only the role:

```
create_profile(
    "Hafsa",
    role="Founder"
)
```

This avoids providing every optional argument.

---

6. Positional + Keyword Arguments

A function call can contain both positional and keyword arguments.

Example:

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

We can write:

```
introduce("Hafsa", age=25, role="Developer")
```

Here:

```
"Hafsa" → positional argument
age=25 → keyword argument
role="Developer" → keyword argument
```

This is valid.

---

7. Important Rule: Positional Arguments Must Come First

This is valid:

```
introduce(
    "Hafsa",
    age=25,
    role="Developer"
)
```

But this is invalid:

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

You cannot put a positional argument after a keyword argument.

Remember:

```
Positional arguments → first
Keyword arguments → after them
```

---

8. You Can Use Only Keyword Arguments

A function can be called entirely with keyword arguments.

```
def product(name, price, quantity):
    print(name, price, quantity)
```

Call:

```
product(
    name="Keyboard",
```

```
    price=5000,
    quantity=2
)
```

This is completely valid.

---

9. Keyword Argument Names Must Match Parameters

Suppose we have:

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

Correct:

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

Incorrect:

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

Python does not know a parameter called `username`.

You would get an error similar to:

```
TypeError: greet() got an unexpected keyword argument 'username'
```

The keyword must match the parameter name.

---

10. Keyword Arguments Are Case Sensitive

Python parameter names are case sensitive.

For example:

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

This works:

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

But this does not:

```
greet(Name="Hafsa")
```

`name` and `Name` are different identifiers.

---

11. Keyword Arguments and Expressions

The value of a keyword argument does not have to be a literal.

It can be a variable:

```
name = "Hafsa"
age = 25

introduce(
    name=name,
    age=age
)
```

It can also be an expression:

```
calculate_total(
    price=1000 * 2
)
```

Python evaluates the expression and passes the resulting value.

---

12. Keyword Arguments With Functions That Return Values

Keyword arguments work normally with functions that return values.

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

We can call:

```
total = calculate_total(
    price=1000,
    quantity=3,
    discount=10
)

print(total)
```

Output:

```
2700.0
```

The parameter names make the calculation easier to understand.

---

13. Real Example: Shopping Cart

Imagine a shopping cart function:

```
def add_product(name, price, quantity=1):
    return {
        "name": name,
        "price": price,
        "quantity": quantity
    }
```

We can write:

```
product = add_product(
    name="Keyboard",
    price=5000,
    quantity=2
)
```

Output:

```
{
    "name": "Keyboard",
    "price": 5000,
    "quantity": 2
}
```

Because `quantity` has a default value, this is also valid:

```
product = add_product(
    name="Keyboard",
    price=5000
)
```

Python automatically uses:

```
quantity = 1
```

---

14. Real Example: haas.dev Resource

Suppose we create a function for generating resource information:

```
def create_resource(title, category="Python", level="Beginner"):
    return {
        "title": title,
```

```
        "category": category,
        "level": level
    }
```

We can use keyword arguments:

```
resource = create_resource(
    title="Keyword Arguments",
    level="Intermediate"
)
```

Python uses:

```
title → "Keyword Arguments"
category → "Python"
level → "Intermediate"
```

The default category remains unchanged.

This is useful when a function has many optional settings.

---

15. Keyword Arguments Improve Configuration Code

Consider:

```
def create_app(name, debug=False, port=8000, host="localhost"):
    pass
```

A positional call might look like:

```
create_app("MyApp", True, 5000, "127.0.0.1")
```

You have to remember what each value means.

Keyword arguments make it clearer:

```
create_app(
    name="MyApp",
    debug=True,
    port=5000,
    host="127.0.0.1"
)
```

This style is common when configuring applications.

---

16. A Common Mistake: Repeating the Same Argument

Consider:

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

This is incorrect:

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

Why?

Python receives two values for `name`:

```
"Hafsa" → name
"Asim" → name
```

This causes an error because one parameter cannot receive two values in this situation.

Use either:

```
greet("Hafsa", age=25)
```

or:

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

---

17. A Common Mistake: Misspelling a Keyword

Suppose:

```
def create_user(name, age):
    pass
```

This is incorrect:

```
create_user(nam="Hafsa", age=25)
```

The parameter is called `name`, not `nam`.

Correct:

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

Python does not automatically guess what you meant.

---

18. A Common Mistake: Forgetting the `=` Sign

Keyword arguments use:

```
parameter=value
```

Correct:

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

Incorrect:

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

The `=` connects the parameter name with the value.

---

19. Keyword Arguments vs Positional Arguments

Feature	Positional	Keyword
Uses position	Yes	No
Uses parameter name	No	Yes
Order matters	Yes	Usually no
Readability	Lower for many arguments	Higher
Can skip optional defaults easily	Less convenient	Very convenient
Example	`greet("Hafsa")`	`greet(name="Hafsa")`

---

20. When Should You Use Keyword Arguments?

Use keyword arguments when:

The function has many parameters

```
create_user(
    name="Hafsa",
    age=25,
    role="Developer",
    active=True
)
```

You want the code to be self-explanatory

```
resize_image(
    width=800,
    height=600
)
```

You want to change a specific default

```
create_resource(
    title="Python Functions",
    level="Advanced"
)
```

The order of values is not obvious

Instead of:

```
configure_app("MyApp", True, 5000, "localhost")
```

use:

```
configure_app(
    name="MyApp",
    debug=True,
    port=5000,
    host="localhost"
)
```

---

21. Positional and Keyword Arguments Together

A useful pattern is:

```
function(required_value, optional_value=...)
```

Example:

```
def calculate_price(price, discount=0, tax=5):
    ...
```

We can call:

```
calculate_price(1000)
```

Or:

```
calculate_price(1000, discount=10)
```

Or:

```
calculate_price(
    1000,
    discount=10,
    tax=8
)
```

This gives us a flexible function interface.

---

22. Keyword Arguments and Function Design

Good function design often uses required parameters for essential information and defaults for common optional settings.

For example:

```
def create_post(title, category="Programming", published=False):
    return {
        "title": title,
        "category": category,
        "published": published
    }
```

`title` is essential.

`category` has a sensible default.

`published` has a safe default.

Now:

```
create_post(
    title="Python Functions"
)
```

is simple.

But we can customize it:

```
create_post(
    title="Python Functions",
    category="Python",
    published=True
)
```

---

23. Keyword Arguments and Readability in Real Projects

In small examples, this:

```
calculate_area(10, 20)
```

may be easy to understand.

But in a larger project, this:

```
calculate_area(
    length=10,
    width=20
)
```

can make the code much easier to maintain.

A future developer does not need to search for the function definition just to remember what `10` and `20` represent.

This is one reason keyword arguments are valuable in real-world Python code.

---

24. Problem Solving Framework

When deciding how to call a function, ask:

Step 1: Are the values obvious from their position?

If yes, positional arguments can be simple.

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

Step 2: Are there many parameters?

If yes, keyword arguments can improve readability.

```
create_user(
    name="Hafsa",
    age=25,
    role="Developer"
)
```

Step 3: Are optional parameters involved?

Keyword arguments make it easy to change only what you need.

```
create_resource(
    title="Python Functions",
    level="Intermediate"
)
```

Step 4: Are positional and keyword arguments mixed?

Always put positional arguments first.

Correct:

```
function("Hafsa", age=25)
```

---

25. Mini Project: User Profile Creator

Create a function that accepts:

- `name` as required
- `age` with a default of `18`
- `role` with a default of `"User"`
- `active` with a default of `True`

```
def create_profile(
    name,
    age=18,
    role="User",
    active=True
):
    return {
        "name": name,
        "age": age,
        "role": role,
        "active": active
    }
```

Now use keyword arguments:

```
profile = create_profile(
    name="Hafsa",
    role="Developer",
    active=True
)

print(profile)
```

This demonstrates both:

- default arguments
- keyword arguments

---

26. 5 Minute Challenge

Create a function:

```
def create_course(title, level="Beginner", lessons=10, published=False):
```

Return a dictionary containing all four values.

Then create these three courses:

Course 1

Only provide the title.

Course 2

Provide the title and level.

Course 3

Use keyword arguments to provide:

```
title
lessons
published
```

without changing the order of the parameters.

Think about how keyword arguments make the third function call easier to understand.

---

27. Exercises

Exercise 1

Create:

```
def greet(name, message="Hello"):
```

Call it using:

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

Then call it using a keyword argument.

Exercise 2

Create:

```
def calculate_bill(amount, tax=5, discount=0):
```

Call the function using only keyword arguments for `tax` and `discount`.

Exercise 3

Create:

```
def create_student(name, grade=9, section="A"):
```

Call it with the arguments in a different order using keywords.

Exercise 4

Create:

```
def create_resource(title, category="Python", level="Beginner"):
```

Create a resource by changing only `level`.

Exercise 5

Find the problem:

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

user_info(name="Hafsa", 25)
```

Fix the function call.

---

28. Mini Quiz

Question 1

What is a keyword argument?

A. An argument passed by position

B. An argument passed using the parameter name

C. A variable created inside a function

D. A default value

Answer: B

Question 2

Which call is valid?

Given:

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

A.

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

B.

```
greet("Hafsa", age=25)
```

C.

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

D. All of the above

Answer: D

Question 3

Which is invalid?

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

A.

```
greet("Hafsa", age=25)
```

B.

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

C.

```
greet(name="Hafsa", 25)
```

D.

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

Answer: C

A positional argument cannot come after a keyword argument.

---

29. Interview Questions

1. What are keyword arguments?

Keyword arguments are arguments passed to a function by explicitly specifying the parameter name.

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

---

2. What is the difference between positional and keyword arguments?

Positional arguments are matched based on their position.

Keyword arguments are matched based on the parameter name.

---

3. Can positional and keyword arguments be used together?

Yes.

```
function("Hafsa", age=25)
```

But positional arguments must come before keyword arguments.

---

4. Can keyword arguments be provided in a different order?

Yes.

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

Python uses the parameter names to match the values.

---

5. What happens if a keyword doesn't match a parameter?

Python raises a `TypeError`.

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

if the function only has a parameter called `name`.

---

6. Can keyword arguments be used with default arguments?

Yes.

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

greet(name="Hafsa")
```

The default value for `message` is still used.

---

30. Cheat Sheet

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

# Positional
greet("Hafsa")

# Keyword
greet(name="Hafsa")

# Both
greet("Hafsa", message="Welcome")

# Keyword arguments in different order
def profile(name, age, role):
    pass

profile(
    role="Developer",
    name="Hafsa",
    age=25
)
```

Important rules

```
parameter=value
```

```
Positional arguments → first
Keyword arguments → after positional arguments
```

```
Keyword name must match the parameter name
```

```
Keyword arguments do not depend on position
```

Key Takeaways

- Keyword arguments pass values using parameter names.
- They make function calls easier to read.
- They allow arguments to be supplied in a different order.
- Positional arguments must come before keyword arguments.
- Keyword names must exactly match the function's parameter names.
- Keyword arguments work especially well with default arguments.
- They are useful when functions have many parameters or optional settings.
- Good use of keyword arguments can make real-world Python code clearer and easier to maintain.

The next step is **Variable Length Arguments:** `*args`, where a function can accept an unknown number of positional arguments.

Visit haas.dev for more resources and guides.

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

Resources:

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