Python `*args` and `**kwargs`

Introduction

So far, we have learned that a function can accept a fixed number of parameters:

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

This function expects two arguments.

But what if we don't know how many values the function will receive?

For example:

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

or:

```
add(10, 20, 30, 40)
```

or:

```
add(10, 20, 30, 40, 50, 60)
```

A normal function cannot accept an unlimited number of positional arguments unless we define enough parameters.

Python solves this problem with:

```
*args
```

and:

```
**kwargs
```

They allow functions to accept a variable number of arguments.

The basic idea is:

```
*args   → multiple positional arguments
**kwargs → multiple keyword arguments
```

---

1. What Does `*args` Mean?

`*args` allows a function to accept any number of positional arguments.

Example:

```
def add_numbers(*args):
    print(args)
```

Now we can pass different numbers of arguments:

```
add_numbers(10, 20)
```

Output:

```
(10, 20)
```

We can pass more:

```
add_numbers(10, 20, 30, 40)
```

Output:

```
(10, 20, 30, 40)
```

Or:

```
add_numbers(5, 10, 15, 20, 25, 30)
```

Output:

```
(5, 10, 15, 20, 25, 30)
```

The function doesn't need to know beforehand how many positional arguments it will receive.

---

2. Why Is It Called `args`?

`args` is simply a common name.

Python does not require the word `args`.

For example:

```
def add_numbers(*numbers):
    print(numbers)
```

This works exactly the same way.

The `*` is what matters.

These are both valid:

```
def function(*args):
    pass
```

```
def function(*values):
    pass
```

However, `*args` is the standard and widely recognized naming convention.

---

3. What Type Is `args`?

Inside the function, `args` is a **tuple**.

Example:

```
def show_values(*args):
    print(type(args))
    print(args)

show_values(10, 20, 30)
```

Output:

```
<class 'tuple'>
(10, 20, 30)
```

So:

```
*args → collects positional arguments into a tuple
```

This is important.

It does not create a list.

---

4. Looping Through `*args`

Because `args` is a tuple, we can loop through it.

```
def show_names(*args):
    for name in args:
        print(name)

show_names("Hafsa", "Asim", "Ali")
```

Output:

```
Hafsa
Asim
Ali
```

This is useful when we don't know how many values will be provided.

---

5. Using `*args` to Add Numbers

We can create a function that adds any number of numbers.

```
def add_numbers(*args):
    total = 0

    for number in args:
        total += number

    return total
```

Now:

```
print(add_numbers(10, 20))
```

Output:

```
30
```

And:

```
print(add_numbers(10, 20, 30, 40))
```

Output:

```
100
```

And:

```
print(add_numbers(5, 10, 15, 20, 25))
```

Output:

```
75
```

The same function works with different numbers of arguments.

---

6. `*args` Can Receive Zero Arguments

A function using `*args` does not have to receive any arguments.

```
def show_values(*args):
    print(args)

show_values()
```

Output:

```
()
```

An empty tuple is created.

We can also check:

```
def show_values(*args):
    if len(args) == 0:
        print("No values were provided")
```

---

7. Accessing Individual `*args` Values

Because `args` is a tuple, we can use indexing.

```
def show_values(*args):
    print(args[0])
    print(args[1])
```

Calling:

```
show_values(10, 20, 30)
```

Output:

```
10
20
```

We can also use slicing:

```
def show_values(*args):
    print(args[1:])
```

Calling:

```
show_values(10, 20, 30, 40)
```

Output:

```
(20, 30, 40)
```

---

8. What Does `**kwargs` Mean?

`**kwargs` allows a function to accept any number of **keyword arguments**.

Example:

```
def show_info(**kwargs):
    print(kwargs)
```

Now:

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

Output:

```
{
    'name': 'Hafsa',
    'age': 25,
    'role': 'Developer'
}
```

Unlike `args`, `kwargs` is a **dictionary**.

So:

```
*args → tuple
**kwargs → dictionary
```

---

9. Why Is It Called `kwargs`?

Again, `kwargs` is a convention.

The `**` is what matters.

This is valid:

```
def show_info(**data):
    print(data)
```

But the standard name is:

```
def show_info(**kwargs):
    print(kwargs)
```

`kwargs` stands for:

```
keyword arguments
```

---

10. Looping Through `**kwargs`

Since `kwargs` is a dictionary, we can use dictionary methods.

```
def show_info(**kwargs):
    for key, value in kwargs.items():
        print(key, "=", value)
```

Calling:

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

Output:

```
name = Hafsa
age = 25
role = Developer
```

---

11. Accessing Individual `**kwargs` Values

Because `kwargs` is a dictionary, we can access values using keys.

```
def show_profile(**kwargs):
    print(kwargs["name"])
```

Calling:

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

Output:

```
Hafsa
```

We can also use `.get()`:

```
def show_profile(**kwargs):
    print(kwargs.get("name"))
```

`.get()` can be safer when a key may not exist.

---

12. `*args` vs `**kwargs`

The easiest way to remember them:

Feature	`*args`	`**kwargs`
Accepts	Positional arguments	Keyword arguments
Stores values as	Tuple	Dictionary
Example	`10, 20, 30`	`name="Hafsa", age=25`
Access	Index	Key
Common use	Unknown number of values	Unknown number of named options

Think:

```
*args  → values
**kwargs → names + values
```

---

13. Using Both `*args` and `**kwargs`

A function can accept both.

```
def show_data(*args, **kwargs):
    print("Positional:", args)
    print("Keyword:", kwargs)
```

Call:

```
show_data(
    10,
    20,
    30,
    name="Hafsa",
    role="Developer"
)
```

Output:

```
Positional: (10, 20, 30)

Keyword: {
    'name': 'Hafsa',
    'role': 'Developer'
}
```

Python separates them automatically.

---

14. How Python Separates Them

Consider:

```
def show_data(*args, **kwargs):
    pass
```

And:

```
show_data(
    10,
    20,
    name="Hafsa",
    age=25
)
```

Python puts:

```
10
20
```

into:

```
args
```

So:

```
args == (10, 20)
```

And:

```
name="Hafsa"
age=25
```

go into:

```
kwargs
```

So:

```
kwargs == {
    "name": "Hafsa",
    "age": 25
}
```

---

15. Regular Parameters With `*args`

We can also have normal parameters.

```
def greet(greeting, *names):
    for name in names:
        print(greeting, name)
```

Call:

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

Output:

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

Here:

```
greeting → "Hello"
names → ("Hafsa", "Asim", "Ali")
```

The regular parameter receives its value first, and the remaining positional arguments are collected into `names`.

---

16. Regular Parameters With `**kwargs`

We can also combine a normal parameter with `**kwargs`.

```
def create_user(name, **details):
    print("Name:", name)
    print("Details:", details)
```

Call:

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

Output:

```
Name: Hafsa
Details: {
    'age': 25,
    'role': 'Developer',
    'country': 'Pakistan'
}
```

Here:

```
name → "Hafsa"
details → dictionary containing the remaining keyword arguments
```

---

17. `*args` With Default Arguments

We can combine `*args` with normal default parameters.

```
def calculate_discount(discount=10, *prices):
    for price in prices:
        print(price - (price * discount / 100))
```

However, this design can sometimes be confusing because positional values are consumed by the regular parameter first.

For example:

```
calculate_discount(20, 1000, 2000)
```

Here:

```
discount → 20
prices → (1000, 2000)
```

When designing functions, make sure the argument structure is clear.

---

18. Keyword Only Arguments

Python also allows us to make parameters keyword-only.

Consider:

```
def create_user(name, *, role="User", active=True):
    print(name, role, active)
```

The `*` here has a different purpose from `*args`.

It means everything after it must be provided using keyword arguments.

This works:

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

But this does not:

```
create_user("Hafsa", "Developer", True)
```

because `role` and `active` are keyword-only parameters.

This technique is useful when we want function calls to be explicit.

---

19. `*args` vs the Standalone `*`

These two forms are different:

`*args`

```
def function(*args):
    pass
```

Means:

> Collect extra positional arguments into a tuple.

Standalone `*`

```
def function(name, *, age):
    pass
```

Means:

> Parameters after `*` must be passed by keyword.

Do not confuse the two.

---

20. Unpacking With `*`

The `*` symbol can also be used when **calling** a function.

Suppose:

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

We have a list:

```
numbers = [10, 20, 30]
```

We can unpack it:

```
result = add(*numbers)
print(result)
```

Output:

```
60
```

Python treats:

```
add(*numbers)
```

as:

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

So `*` can collect values in a function definition and unpack values in a function call.

---

21. Unpacking With `**`

`**` can also unpack a dictionary into keyword arguments.

Suppose:

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

We have:

```
profile = {
    "name": "Hafsa",
    "age": 25,
    "role": "Developer"
}
```

We can call:

```
create_profile(**profile)
```

Python treats it like:

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

This is called **dictionary unpacking**.

---

22. Collecting vs Unpacking

This is one of the most important concepts.

In a function definition

```
def function(*args):
    pass
```

`*args` **collects** multiple positional arguments.

```
def function(**kwargs):
    pass
```

`**kwargs` **collects** multiple keyword arguments.

In a function call

```
function(*values)
```

`*` **unpacks** an iterable.

```
function(**data)
```

`**` **unpacks** a dictionary.

So:

```
Definition → collect
Call → unpack
```

---

23. Real Example: haas.dev Resource Creator

Suppose we want a flexible function that creates resource data.

```
def create_resource(title, *tags, **details):
    return {
        "title": title,
        "tags": tags,
        "details": details
    }
```

Now:

```
resource = create_resource(
    "Python Functions",
    "python",
    "functions",
    "beginner",
    level="Beginner",
    category="Programming"
)
```

The result is conceptually:

```
{
    "title": "Python Functions",
    "tags": (
        "python",
        "functions",
        "beginner"
    ),
    "details": {
        "level": "Beginner",
        "category": "Programming"
    }
}
```

This is useful when a function needs to accept flexible metadata.

---

24. Real Example: Shopping Cart

Suppose we want to calculate the total of any number of products.

```
def calculate_total(*prices):
    total = 0

    for price in prices:
```

```
        total += price

    return total
```

Now:

```
print(calculate_total(500, 1000))
```

Output:

```
1500
```

And:

```
print(calculate_total(500, 1000, 750, 1200))
```

Output:

```
3450
```

The function doesn't need a fixed number of price parameters.

---

25. Real Example: Flexible User Settings

Suppose we want to create a user settings function.

```
def create_settings(**settings):
    return settings
```

Now:

```
settings = create_settings(
    theme="dark",
    notifications=True,
    language="English",
    font_size=16
)
```

The result is:

```
{
    "theme": "dark",
    "notifications": True,
    "language": "English",
    "font_size": 16
}
```

The function can accept new settings without changing its parameter list.

---

26. Why Frameworks Use `**kwargs`

You will frequently see `**kwargs` in Python libraries and frameworks.

A library may provide a function that supports many optional settings.

Instead of creating dozens of fixed parameters, it can accept additional keyword arguments.

Conceptually:

```
def configure_component(**kwargs):
    pass
```

Then users can provide:

```
configure_component(
    color="blue",
    size="large",
    enabled=True
)
```

This makes APIs flexible.

---

27. Why `*args` Is Useful in Real Projects

Imagine a logging function:

```
def log_messages(*messages):
    for message in messages:
        print(message)
```

Now:

```
log_messages(
    "Application started",
    "User logged in",
    "Database connected"
)
```

The function can handle any number of messages.

Another example is a mathematical function:

```
def maximum(*numbers):
    return max(numbers)
```

Now:

```
print(maximum(10, 50, 20, 80, 30))
```

Output:

```
80
```

---

28. Common Mistake: Thinking `args` Is a List

Consider:

```
def show_values(*args):
    print(type(args))
```

The result is:

```
<class 'tuple'>
```

Not:

```
<class 'list'>
```

So this:

```
args.append(10)
```

will not work because tuples do not have `.append()`.

If you need a list:

```
values = list(args)
```

Now `values` is a list.

---

29. Common Mistake: Thinking `kwargs` Is a Tuple

`kwargs` is the opposite.

```
def show_info(**kwargs):
    print(type(kwargs))
```

Output:

```
<class 'dict'>
```

So we can use dictionary operations:

```
kwargs["name"]
```

```
kwargs.items()
```

```
kwargs.get("name")
```

---

30. Common Mistake: Forgetting the `*`

This:

```
def add_numbers(args):
    pass
```

is not the same as:

```
def add_numbers(*args):
    pass
```

Without `*`, `args` is simply one normal parameter.

With `*`, Python can collect multiple positional arguments.

Similarly:

```
def create_user(kwargs):
    pass
```

accepts one normal argument.

But:

```
def create_user(**kwargs):
    pass
```

accepts multiple keyword arguments.

---

31. Common Mistake: Using `**kwargs` With Positional Arguments

This function:

```
def show_info(**kwargs):
    print(kwargs)
```

expects keyword arguments.

So:

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

works.

But:

```
show_info("Hafsa", 25)
```

does not work because those are positional arguments.

If you want both types:

```
def show_info(*args, **kwargs):
    print(args)
    print(kwargs)
```

---

32. Common Mistake: Unexpected Keyword Names

Consider:

```
def create_user(**kwargs):
    print(kwargs)
```

This function accepts any keyword names.

But if we have:

```
def greet(name, **kwargs):
    print(name)
```

then:

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

works because `age` goes into `kwargs`.

This flexibility is powerful, but it also means you should validate values when your application requires specific options.

---

33. `*args` and `**kwargs` With Normal Parameters

A common function structure is:

```
def function(required, *args, **kwargs):
    pass
```

Example:

```
def process(name, *items, **options):
    print("Name:", name)
```

```
    print("Items:", items)
    print("Options:", options)
```

Call:

```
process(
    "Hafsa",
    "Python",
    "JavaScript",
    level="Intermediate",
    active=True
)
```

Conceptually:

```
name → "Hafsa"

items → (
    "Python",
    "JavaScript"
)

options → {
    "level": "Intermediate",
    "active": True
}
```

This gives the function a very flexible interface.

---

34. Parameter Order

When using all these types together, the order matters.

A common structure is:

```
def function(
    normal_parameter,
    *args,
    keyword_parameter,
    **kwargs
):
    pass
```

For example:

```
def example(name, *args, role="User", **kwargs):
    print(name)
    print(args)
    print(role)
    print(kwargs)
```

The exact design depends on what the function needs, but understanding the order prevents confusing errors.

---

35. When Should You Use `*args`?

Use `*args` when:

- The number of positional values is unknown.
- All the values represent a similar type of information.
- You want a flexible function.
- You are writing a wrapper around another function.
- A function naturally accepts multiple values.

Example:

```
def calculate_total(*prices):
    return sum(prices)
```

---

36. When Should You Use `**kwargs`?

Use `**kwargs` when:

- The number of keyword options is unknown.
- The options are naturally represented as named values.
- You are creating configurable functions.
- You are building flexible APIs.
- You are passing optional settings to another function.

Example:

```
def configure(**settings):
    return settings
```

---

37. When Should You NOT Use Them?

Don't use `*args` or `**kwargs` simply because they exist.

If a function always needs exactly three values:

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

this is clearer than:

```
def create_user(*args):
    pass
```

Explicit parameters communicate what the function actually needs.

Use variable-length arguments when flexibility is genuinely useful.

---

38. Problem Solving Framework

Before using `*args` or `**kwargs`, ask:

Step 1: Is the number of values fixed?

If yes:

```
def add(a, b):
    pass
```

Use normal parameters.

Step 2: Can the number of positional values change?

If yes:

```
def add(*numbers):
    pass
```

Use `*args`.

Step 3: Can the number of named options change?

If yes:

```
def configure(**options):
    pass
```

Use `**kwargs`.

Step 4: Do you need both?

Use:

```
def function(*args, **kwargs):
    pass
```

Step 5: Is flexibility actually helping?

If normal parameters make the function clearer, use normal parameters instead.

---

39. Mini Project: Flexible Calculator

Create a calculator that can perform operations on any number of values.

```
def calculate(operation, *numbers):
    if operation == "sum":
        return sum(numbers)

    if operation == "max":
        return max(numbers)

    if operation == "min":
        return min(numbers)

    return "Unknown operation"
```

Use it:

```
print(calculate("sum", 10, 20, 30))
```

Output:

```
60
```

And:

```
print(calculate("max", 10, 50, 20, 80))
```

Output:

```
80
```

And:

```
print(calculate("min", 10, 50, 20, 80))
```

Output:

```
10
```

The number of values can change without changing the function definition.

---

40. Mini Project: Flexible Resource Creator

Create a function for haas.dev resources:

```
def create_resource(title, *tags, **details):
    resource = {
        "title": title,
        "tags": tags,
        "details": details
    }

    return resource
```

Use it:

```
resource = create_resource(
    "Python Functions",
    "python",
    "functions",
    "programming",
    level="Beginner",
    category="Python",
    published=True
)

print(resource)
```

This function can support different numbers of tags and additional metadata without changing its definition.

---

41. 5 Minute Challenge

Create a function:

```
def create_profile(name, *skills, **details):
```

Requirements:

- `name` should be required.
- Any number of skills should be accepted.
- Additional profile information should be accepted as keyword arguments.
- Return everything as a dictionary.

Try:

```
profile = create_profile(
    "Hafsa",
    "Python",
```

```
    "JavaScript",
    "React",
    role="Developer",
    experience=2,
    active=True
)
```

Your result should contain:

```
name
skills
role
experience
active
```

---

42. Exercises

Exercise 1

Create:

```
def add_numbers(*numbers):
```

Return the sum of all numbers.

Exercise 2

Create:

```
def show_students(*names):
```

Print every student name.

Exercise 3

Create:

```
def create_settings(**settings):
```

Return the settings dictionary.

Test it with:

```
create_settings(
    theme="dark",
    language="English",
```

```
    notifications=True
)
```

Exercise 4

Create:

```
def student(name, *subjects, **details):
```

Store:

- name
- subjects
- additional details

in a dictionary.

Exercise 5

Given:

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

Create a dictionary:

```
user = {
    "name": "Hafsa",
    "age": 25,
    "role": "Developer"
}
```

Call the function using:

```
**user
```

---

43. Mini Quiz

Question 1

What does `*args` collect?

A. Keyword arguments

B. Positional arguments

C. Dictionaries

D. Functions

Answer: B

Question 2

What type is `args` inside the function?

A. List

B. Dictionary

C. Tuple

D. String

Answer: C

Question 3

What type is `kwargs`?

A. Tuple

B. List

C. Dictionary

D. Set

Answer: C

Question 4

What does this function accept?

```
def function(**kwargs):
    pass
```

A. Any number of positional arguments

B. Any number of keyword arguments

C. Exactly two arguments

D. Only strings

Answer: B

Question 5

What does this do?

```
numbers = [10, 20, 30]
add(*numbers)
```

Answer:

It unpacks the list so its values are passed as separate positional arguments.

Conceptually:

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

---

44. Interview Questions

1. What are `*args` and `**kwargs`?

`*args` allows a function to accept a variable number of positional arguments.

`**kwargs` allows a function to accept a variable number of keyword arguments.

---

2. What type is `args`?

`args` is a tuple.

---

3. What type is `kwargs`?

`kwargs` is a dictionary.

---

4. Can `*args` and `**kwargs` be used together?

Yes.

```
def function(*args, **kwargs):
    pass
```

---

5. What is the difference between `*args` and `**kwargs`?

```
*args → positional → tuple
**kwargs → keyword → dictionary
```

---

6. What does `*` do when used in a function call?

It unpacks an iterable into positional arguments.

```
numbers = [1, 2, 3]

function(*numbers)
```

---

7. What does `**` do when used in a function call?

It unpacks a dictionary into keyword arguments.

```
data = {
    "name": "Hafsa",
    "age": 25
}
```

```
function(**data)
```

---

8. Are `args` and `kwargs` required names?

No.

These are conventions.

For example:

```
def function(*values, **options):
    pass
```

is valid.

The `*` and `**` are what provide the behavior.

---

45. Cheat Sheet

`*args`

```
def show_values(*args):
    print(args)

show_values(10, 20, 30)
```

Result:

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

`args` is a tuple.

`**kwargs`

```
def show_info(**kwargs):
    print(kwargs)

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

Result:

```
{
    "name": "Hafsa",
```

```
    "age": 25
}
```

`kwargs` is a dictionary.

Both

```
def function(*args, **kwargs):
    pass
```

Unpack a list

```
numbers = [10, 20, 30]

function(*numbers)
```

Unpack a dictionary

```
data = {
    "name": "Hafsa",
    "age": 25
}

function(**data)
```

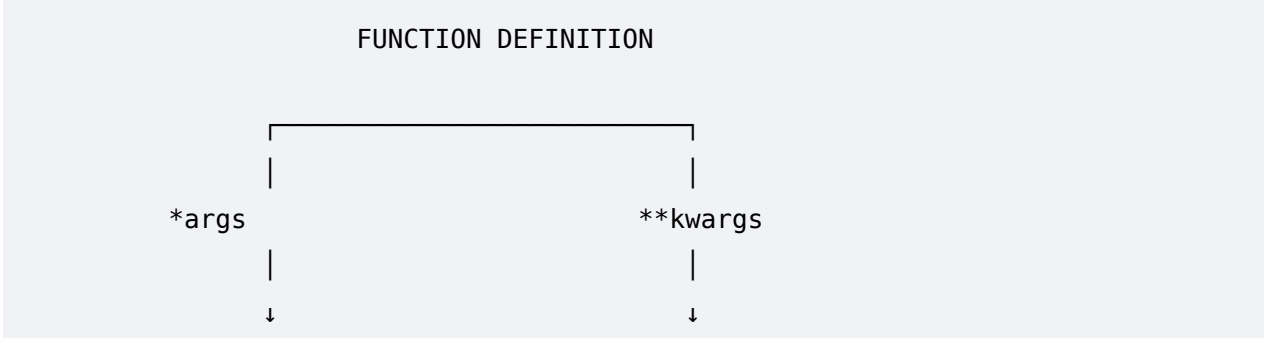
Keyword only

```
def function(name, *, age):
    pass
```

`age` must be provided as a keyword argument:

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

`*args` vs `**kwargs` at a Glance



```
    Positional arguments        Keyword arguments
              |                           |
              ↓                           ↓
           Tuple                      Dictionary
```

Example:

```
def profile(*args, **kwargs):
    print(args)
    print(kwargs)
```

Call:

```
profile(
    "Hafsa",
    "Python",
    role="Developer",
    level="Intermediate"
)
```

Result:

```
args
→ ("Hafsa", "Python")

kwargs
→ {
    "role": "Developer",
    "level": "Intermediate"
}
```

Key Takeaways

- `*args` accepts any number of positional arguments.
- `**kwargs` accepts any number of keyword arguments.
- `args` is stored as a tuple.
- `kwargs` is stored as a dictionary.
- `*args` is useful when the number of positional values is unknown.
- `**kwargs` is useful when the number of named options is unknown.
- Both can be used together.
- `*` can unpack lists, tuples, and other iterables when calling a function.
- `**` can unpack dictionaries into keyword arguments.

- `*` by itself can make parameters keyword-only.
- `args` and `kwargs` are conventional names, not required names.
- Do not use `*args` or `**kwargs` when explicit parameters would make the function clearer.

The next step is **Scope in Python**, where we learn where variables can be accessed inside and outside functions.

Visit haas.dev for more resources and guides.

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

Resources:

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