

# Args and Kwargs Explained

Explain \*args and \*\*kwargs in python

In Python, `*args` and `**kwargs` are used to handle variable numbers of arguments in function definitions.

## 1. `*args` (Non-Keyword Arguments)

- `*args` allows a function to accept any number of **positional arguments** as a tuple.
- Useful when you don't know in advance how many arguments will be passed.

Example:

```
python

def sum_all(*args):
    return sum(args)

print(sum_all(1, 2, 3, 4))  # Output: 10
print(sum_all(10, 20))     # Output: 30
```

Here, `args` is a tuple that collects all the positional arguments passed to the function.

---

## 2. `**kwargs` (Keyword Arguments)

- `**kwargs` allows a function to accept any number of **keyword arguments** as a dictionary.
- Useful when you need to handle named arguments dynamically.

Example:

```
python

def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

```
print_info(name="Alice", age=25, country="USA")
```

### Output:

```
makefile
```

```
name: Alice  
age: 25  
country: USA
```

Here, `kwargs` is a dictionary that stores the keyword arguments.

---

## 3. Using `*args` and `**kwargs` Together

You can use both `*args` and `**kwargs` in the same function.

### Example:

```
python
```

```
def func(a, b, *args, **kwargs):  
    print(f"a: {a}, b: {b}")  
    print(f"args: {args}")  
    print(f"kwargs: {kwargs}")  
  
func(1, 2, 3, 4, x=5, y=6)
```

### Output:

```
yaml
```

```
a: 1, b: 2  
args: (3, 4)  
kwargs: {'x': 5, 'y': 6}
```

## 4. Unpacking `*args` and `**kwargs`

You can use `*` and `**` to **unpack** arguments when calling a function.

**Example:**

python

```
def greet(name, age):  
    print(f"Hello, my name is {name} and I am {age} years old.")  
  
person_info = ("John", 30)  
greet(*person_info) # Unpacking tuple  
  
details = {"name": "Jane", "age": 28}  
greet(**details) # Unpacking dictionary
```

## Summary

Feature	<code>*args</code>	<code>**kwargs</code>
Collects	Positional arguments (Tuple)	Keyword arguments (Dictionary)
Example Call	<code>func(1, 2, 3)</code>	<code>func(name="Alice", age=25)</code>
Unpacking	<code>func(*my_tuple)</code>	<code>func(**my_dict)</code>

Let me know if you need more examples! 🚀

↳ name: Alice age: 25 country: USA

why are the quotes removed from here in the values

Great observation! The reason the quotes are removed in the output:

makefile

```
name: Alice  
age: 25  
country: USA
```

is because of how Python's `print()` function handles strings.

## Explanation:

### 1. Strings don't need explicit quotes when printed

- When you print a string normally, Python outputs it without quotes.
- Example:

```
python

name = "Alice"
print(name)  # Output: Alice
```

### 2. Inside `print_info(**kwargs)`, `kwargs.items()` gives key-value pairs

- `for key, value in kwargs.items():` iterates over the dictionary.
- `print(f"{key}: {value}")` prints the values directly.
- Since `value` is a string ( `"Alice"` ), Python prints it without quotes.

### 3. If you explicitly want quotes, you can modify the print statement:

```
python

def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {repr(value)}")  # Using repr() to show quotes

print_info(name="Alice", age=25, country="USA")
```

## Output:

```
vbnet

name: 'Alice'
age: 25
country: 'USA'
```

- `repr(value)` ensures strings are printed with quotes.
- Integers ( `25` ) remain unquoted.

Would you like more tweaks to format the output differently? 🚀

```
def greet(name, age): print(f"Hello, my name is {name} and I am {age} years old.") person_info =  
↳ ("John", 30) greet(*person_info) \# Unpacking tuple details = {"name": "Jane", "age": 28}  
greet(**details) \# Unpacking dictionary
```

what will be its output ?

Let's analyze the code and predict its output.

## Code:

python

```
def greet(name, age):  
    print(f"Hello, my name is {name} and I am {age} years old.")  
  
person_info = ("John", 30)  
greet(*person_info)  # Unpacking tuple  
  
details = {"name": "Jane", "age": 28}  
greet(**details)  # Unpacking dictionary
```

## Execution:

### 1. `greet(*person_info)`

- `person_info = ("John", 30)` (a tuple)
- `*person_info` unpacks this tuple, passing `"John"` as `name` and `30` as `age`.
- The function call becomes:

python

```
greet("John", 30)
```

- Output:

pgsql

```
Hello, my name is John and I am 30 years old.
```

### 2. `greet(**details)`

- `details = {"name": "Jane", "age": 28}` (a dictionary)
- `**details` unpacks the dictionary, passing `"Jane"` as `name` and `28` as `age`.

- The function call becomes:

```
python

greet(name="Jane", age=28)
```

- Output:

```
pgsql

Hello, my name is Jane and I am 28 years old.
```

## Final Output:

```
pgsql

Hello, my name is John and I am 30 years old.
Hello, my name is Jane and I am 28 years old.
```

Would you like to test this with different values? 🚀

↳ name

what if here instead of name we would have written "NAME" ?

If you modify the dictionary key from "name" to "NAME" like this:

```
python

details = {"NAME": "Jane", "age": 28}
greet(**details)  # Unpacking dictionary
```

## What Happens?

- The `greet` function expects the parameter names to be **exactly** `name` and `age` :

```
python

def greet(name, age):
```

- But when using `**details` , Python tries to match dictionary keys to function parameter names.

- Since `"NAME"` (uppercase) does **not** match `name` (lowercase), Python raises an error.

## Error Output:

CSS

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

## How to Fix?

1. Use the correct key ( `"name"` ) in the dictionary:

python

```
details = {"name": "Jane", "age": 28}
greet(**details) # ✅ Works fine
```

2. Modify the function to accept `"NAME"` :

python

```
def greet(NAME, age):
    print(f"Hello, my name is {NAME} and I am {age} years old.")

details = {"NAME": "Jane", "age": 28}
greet(**details) # ✅ Works fine now
```

Would you like to test other cases? 🚀

what are the difference between positional and keyword argument ?

## Positional vs Keyword Arguments in Python

In Python, function arguments can be passed in two ways:

1. Positional Arguments
2. Keyword Arguments



---

### 1. Positional Arguments

- Arguments are passed based on their **position** in the function definition.
- The order **matters**.


### Example:

python

```
def greet(name, age):  
    print(f"Hello, my name is {name} and I am {age} years old.")  
  
greet("Alice", 25) #  Correct  
greet(25, "Alice") #  Incorrect order, changes meaning
```

### Output:

pgsql

```
Hello, my name is Alice and I am 25 years old.  
Hello, my name is 25 and I am Alice years old.  (Incorrect meaning)
```

**Issue:** If the order is wrong, incorrect values are assigned.

## 2. Keyword Arguments

- Arguments are passed using the **parameter names**.
- The order **does not matter**.
- Improves **readability** and reduces **errors**.

### Example:


python

```
greet(name="Alice", age=25) #  Correct  
greet(age=25, name="Alice") #  Correct (Order doesn't matter)
```

### Output:

pgsql



Hello, my name is Alice and I am 25 years old.  
Hello, my name is Alice and I am 25 years old.  (Correct output)


**Benefit:** Even if we change the order, it works correctly.

## Key Differences


Feature	Positional Arguments	Keyword Arguments
Order	Must be in correct order	Can be in any order
Readability	Less readable for long argument lists	More readable
Flexibility	Must remember parameter positions	No need to remember positions
Error Risk	High if order is wrong	Low, since names are explicit
Mixing Allowed?	Yes, but <b>positional must come first</b>	Yes

## Mixing Positional & Keyword Arguments

 **Allowed:**

```
python
greet("Alice", age=25) #  Positional + Keyword
```

 **Not Allowed:**

```
python
greet(name="Alice", 25) #  SyntaxError (Positional after Keyword)
```

**Rule:** Positional arguments must always come first before keyword arguments.

---

## Best Practice

- Use **positional arguments** when the function has only **1-2 parameters**.
- Use **keyword arguments** for functions with **many parameters** for better readability.
- Use **both** when it makes sense.

Would you like to see more examples? 