Python Default Arguments

Introduction

When we create a function, some parameters may have values that are used most of the time. Instead of asking the user to provide those values every time, Python allows us to give a parameter a **default value**.

These are called **default arguments**.

For example:

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

Here:

- `name` is a required parameter.
- `message` has a default value of `"Hello"`.

So we can call the function in two ways:

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

Output:

```
Hello Hafsa
```

Or provide a different value:

```
greet("Hafsa", "Welcome")
```

Output:

```
Welcome Hafsa
```

The important idea is:

> A default argument gives a parameter a value that Python will use automatically when no value is provided.

---

1. Why Do We Need Default Arguments?

Imagine we create a function that calculates the price of a product after applying a discount.

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

Every time we call it, we must provide a discount:

```
calculate_price(1000, 10)
calculate_price(2000, 10)
calculate_price(3000, 10)
```

But what if our store normally gives a 10% discount?

We can make `10` the default value:

```
def calculate_price(price, discount=10):
    return price - (price * discount / 100)
```

Now:

```
calculate_price(1000)
```

automatically uses a 10% discount.

Output:

```
900.0
```

But we can still provide another discount:

```
calculate_price(1000, 20)
```

Output:

```
800.0
```

So default arguments make functions more flexible.

---

2. Basic Syntax

The syntax is:

```
def function_name(parameter=default_value):
    # function body
```

Example:

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

Calling without an argument:

```
greet()
```

Output:

```
Hello Guest
```

Calling with an argument:

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

Output:

```
Hello Hafsa
```

---

3. Required Parameters vs Default Parameters

A function can contain both.

```
def student_info(name, grade=9):
    print("Name:", name)
    print("Grade:", grade)
```

Here:

```
name
```

is required.

```
grade
```

has a default value.

So this works:

```
student_info("Hafsa")
```

Output:

```
Name: Hafsa
Grade: 9
```

And this also works:

```
student_info("Hafsa", 10)
```

Output:

```
Name: Hafsa
Grade: 10
```

---

4. How Python Decides Which Value to Use

Python follows a simple rule:

> If the caller provides a value, Python uses that value. Otherwise, Python uses the default value.

Example:

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

No argument:

```
greet()
```

Python uses:

```
Guest
```

Argument provided:

```
greet("Asim")
```

Python uses:

```
Asim
```

The default value is therefore a fallback.

---

5. Multiple Default Arguments

A function can have more than one default parameter.

```
def profile(name="Guest", age=18, city="Gujranwala"):
    print(name)
    print(age)
    print(city)
```

Calling:

```
profile()
```

Output:

```
Guest
18
Gujranwala
```

We can also change specific values:

```
profile("Hafsa", 25, "Lahore")
```

Output:

```
Hafsa
25
Lahore
```

---

6. Required Parameters Must Come Before Default Parameters

This is an important Python rule.

Correct:

```
def student(name, grade=9):
    print(name, grade)
```

Incorrect:

```
def student(grade=9, name):
    print(name, grade)
```

Python will produce a syntax error.

Why?

Because Python needs to know how to match positional arguments clearly.

The general rule is:

```
required parameters → default parameters
```

For example:

```
def function(a, b, c=10, d=20):
    pass
```

This is valid.

But:

```
def function(a=10, b, c):
    pass
```

is invalid.

---

7. Default Arguments With Positional Arguments

Consider:

```
def introduce(name, role="Developer"):
    print(name, "is a", role)
```

We can provide only the required argument:

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

Output:

```
Hafsa is a Developer
```

Or provide both:

```
introduce("Hafsa", "Founder")
```

Output:

```
Hafsa is a Founder
```

---

8. Default Arguments With Keyword Arguments

Default arguments become even more useful when combined with keyword arguments.

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

We can change only `country`:

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

Output:

```
Hafsa
Developer
Canada
```

The other default value remains unchanged.

This is useful when a function has several optional settings.

---

9. Default Arguments Are Optional

A default parameter is not always required.

For example:

```
def connect_database(host="localhost"):
    print("Connecting to", host)
```

We can simply write:

```
connect_database()
```

Output:

```
Connecting to localhost
```

Or:

```
connect_database("production-server")
```

Output:

```
Connecting to production-server
```

This pattern is common in real applications where there is a normal or common configuration.

---

10. Different Types of Default Values

Default values are not limited to numbers.

They can be strings:

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

Booleans:

```
def login(is_admin=False):
    print(is_admin)
```

Numbers:

```
def calculate_tax(rate=5):
    print(rate)
```

`None`:

```
def search(query=None):
    print(query)
```

The type depends on what makes sense for the function.

---

11. Using `None` as a Default

`None` is often used when we want to represent "no value was provided."

Example:

```
def search_user(name=None):
    if name is None:
        print("No search term provided")
    else:
        print("Searching for:", name)
```

Calling:

```
search_user()
```

Output:

```
No search term provided
```

Calling:

```
search_user("Hafsa")
```

Output:

```
Searching for: Hafsa
```

This is a very common pattern in Python.

---

12. Default Arguments and Return Values

Default arguments can work together with return values.

```
def calculate_area(length, width=10):
    return length * width
```

Now:

```
area = calculate_area(5)
print(area)
```

Output:

```
50
```

The default width was `10`.

We can override it:

```
area = calculate_area(5, 20)
print(area)
```

Output:

```
100
```

---

13. Real Example: Student Result

Suppose we create a function that calculates a student's final score.

```
def calculate_result(marks, bonus=5):
    return marks + bonus
```

If the student receives no special bonus:

```
print(calculate_result(80))
```

Output:

```
85
```

If the bonus is different:

```
print(calculate_result(80, 10))
```

Output:

```
90
```

The function has a normal default behavior while still allowing customization.

---

14. Real Example: haas.dev Resource Data

Suppose haas.dev has a function that creates information for a resource.

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

Now we can create a normal Python resource:

```
resource = create_resource("Python Functions")
print(resource)
```

Output:

```
{
    'title': 'Python Functions',
    'category': 'Python',
    'level': 'Beginner'
}
```

We can change the category or level when needed:

```
resource = create_resource(
    "Python Return Values",
    level="Intermediate"
)
```

Output:

```
{
    'title': 'Python Return Values',
    'category': 'Python',
    'level': 'Intermediate'
}
```

The defaults provide sensible values without forcing us to type them every time.

---

15. Default Arguments Reduce Repetition

Without defaults:

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

create_user("Hafsa", "Pakistan", "Developer")
create_user("Asim", "Pakistan", "Developer")
create_user("Ali", "Pakistan", "Developer")
```

We repeatedly provide the same values.

With defaults:

```
def create_user(name, country="Pakistan", role="Developer"):
    pass
```

Now:

```
create_user("Hafsa")
create_user("Asim")
create_user("Ali")
```

The repeated information can be defined once.

---

16. Overriding a Default Value

"Default" does not mean "fixed."

It simply means:

> Use this value when the caller doesn't provide another value.

Example:

```
def download_file(format="pdf"):
    print("Downloading as", format)
```

Default:

```
download_file()
```

Output:

```
Downloading as pdf
```

Override:

```
download_file("docx")
```

Output:

```
Downloading as docx
```

---

17. A Common Mistake: Forgetting the Required Argument

Consider:

```
def calculate_total(price, tax=5):
    return price + (price * tax / 100)
```

Here `price` is required.

This will cause an error:

```
calculate_total()
```

because no `price` was provided.

But this works:

```
calculate_total(1000)
```

The result uses the default tax rate of `5`.

---

18. A Common Mistake: Wrong Parameter Order

Incorrect:

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

Correct:

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

Remember:

```
Required → Default
```

not:

```
Default → Required
```

---

19. Mutable Default Arguments

There is an important Python issue with mutable default values such as lists and dictionaries.

Avoid code like this unless you specifically understand the behavior:

```
def add_item(item, items=[]):
    items.append(item)
    return items
```

The list can be reused between function calls because the default object is created once.

For example:

```
print(add_item("Python"))
print(add_item("JavaScript"))
```

This may produce:

```
['Python']
['Python', 'JavaScript']
```

Many beginners expect the second call to start with an empty list.

A safer pattern is:

```
def add_item(item, items=None):
    if items is None:
        items = []

    items.append(item)
    return items
```

Now every call without an `items` argument gets a new list.

This is an important best practice.

---

20. Why Does This Happen?

Default parameter values are evaluated when the function is defined, not every time the function is called.

For immutable values such as:

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

this usually behaves exactly as expected.

But mutable objects such as lists can be changed:

```
def add_item(item, items=[]):
    items.append(item)
```

The same default list can therefore be modified and reused.

That is why `None` is commonly used for optional lists and dictionaries.

---

21. Default Arguments vs Normal Arguments

Normal Parameter	**Default Parameter**
Usually requires a value	Value is optional

No automatic fallback	Has a fallback value
`def greet(name):`	`def greet(name="Guest"):`
Caller must provide value	Caller can omit value
Good for required information	Good for optional/common settings

---

22. Default Arguments vs Hardcoded Values

Consider:

```
def calculate_discount(price):
    discount = 10
    return price - (price * discount / 100)
```

The discount cannot easily be changed from outside.

With a default parameter:

```
def calculate_discount(price, discount=10):
    return price - (price * discount / 100)
```

Now the normal behavior is still 10%:

```
calculate_discount(1000)
```

But we can customize it:

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

This makes the function reusable.

---

23. Problem Solving Framework

When deciding whether a parameter should have a default value, ask:

Step 1: Is the value required?

If yes, make it a normal parameter.

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

Step 2: Is there a common value?

If yes, consider making it the default.

```
def create_user(name, country="Pakistan"):
    pass
```

Step 3: Should the caller still be able to change it?

If yes, a default parameter is useful.

```
create_user("Hafsa")
create_user("Hafsa", "Canada")
```

Step 4: Could the default be a mutable object?

If yes, consider using `None`.

```
def create_list(items=None):
    if items is None:
        items = []
```

---

24. Mini Project: Product Price Calculator

Create a function that calculates the final price of a product.

Requirements:

- `price` should be required.
- `discount` should default to `0`.
- `tax` should default to `5`.
- Return the final price.

Example:

```
def final_price(price, discount=0, tax=5):
    price = price - (price * discount / 100)
```

```
    price = price + (price * tax / 100)
    return price
```

Test it:

```
print(final_price(1000))
print(final_price(1000, 10))
print(final_price(1000, 10, 8))
```

The function now supports different situations without requiring unnecessary arguments every time.

---

25. 5 Minute Challenge

Create a function called:

```
create_account()
```

It should accept:

```
username
role = "User"
active = True
```

The function should return a dictionary.

Example:

```
create_account("Hafsa")
```

Expected structure:

```
{
    "username": "Hafsa",
    "role": "User",
    "active": True
}
```

Then try:

```
create_account("Hafsa", "Admin", False)
```

Think about which values are required and which values can have sensible defaults.

---

26. Exercises

Exercise 1

Create a function:

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

Print a greeting.

Test it with and without an argument.

Exercise 2

Create:

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

Return the amount after adding tax.

Exercise 3

Create:

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

Return the student's information.

Exercise 4

Create:

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

Return a dictionary containing the post information.

Exercise 5

Create a function using `None` safely:

```
def add_skill(skill, skills=None):
```

Add the skill to the list and return the list.

Make sure separate function calls don't accidentally share the same list.

---

27. Mini Quiz

Question 1

What is a default argument?

A. A parameter that must always be provided

B. A parameter with a fallback value

C. A variable inside a function

D. A return value

Answer: B

Question 2

Which function definition is valid?

A.

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

B.

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

C.

```
def greet(name=, message):
    pass
```

D.

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

Answer: B

Question 3

What happens here?

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

greet()
```

Answer:

```
Guest
```

Python uses the default value because no argument was provided.

---

28. Interview Questions

1. What are default arguments in Python?

Default arguments are parameters that already have a value. If the caller does not provide a value, Python uses the default.

---

2. Can a function have both required and default parameters?

Yes.

```
def calculate(price, tax=5):
    pass
```

---

3. Where should default parameters appear?

After required parameters.

```
def function(required, optional=10):
    pass
```

---

4. Can a default value be overridden?

Yes.

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

greet("Hafsa")
```

The provided value replaces the default for that call.

---

5. Why is `None` commonly used as a default for lists?

Because using a mutable list directly as a default can cause the same list object to be reused between calls.

Safer:

```
def add_item(item, items=None):
    if items is None:
        items = []
```

---

29. Cheat Sheet

```
# Basic default
def greet(name="Guest"):
    print(name)
```

```
# Required + default
def student(name, grade=9):
    print(name, grade)

# Multiple defaults
def profile(name="Guest", age=18):
    print(name, age)

# Override default
profile("Hafsa", 25)

# Keyword argument
profile(age=25)

# None as safe default
def add_item(item, items=None):
    if items is None:
        items = []

    items.append(item)
    return items
```

Remember

```
Default argument = fallback value
```

```
Provided value → use provided value
No value → use default
```

```
Required parameters → first
Default parameters → after them
```

```
Mutable default → prefer None
```

Key Takeaways

- A default argument gives a parameter a fallback value.
- Default parameters are optional when calling the function.
- The caller can override a default value.
- Required parameters must come before default parameters.
- Default arguments make functions more flexible and reusable.
- `None` is commonly used when an optional value should start as empty.

- Be careful with mutable default arguments such as lists and dictionaries.
- Default arguments are especially useful for common settings and optional configuration.

The next step is **Keyword Arguments**, where you can provide function arguments by parameter name instead of relying only on their position.

Visit haas.dev for more resources and guides.

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

Resources:

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