**Decorators in Python**

A **decorator** in Python is a design pattern that allows you to **modify the behavior of a function or class method** without changing its code. Decorators are used to add functionality to an existing function or method in a clean and readable way.

In Python, decorators are implemented as higher-order functions, meaning that they are functions that take another function as an argument and extend or alter its behavior before returning it.

**How Decorators Work**

A decorator wraps a function, allowing you to add code that runs **before** or **after** the function. Decorators are commonly used for:

* Logging
* Access control and authentication
* Timing function execution
* Caching

**Syntax for Decorators**

The syntax for decorators in Python is the @decorator\_name placed above the function definition. This is syntactic sugar for function = decorator(function).

**Example: Basic Decorator**

Let’s start with a simple example to demonstrate how decorators work.

python

Copy code

# Define a decorator

def my\_decorator(func):

def wrapper():

print("Something before the function.")

func() # Call the original function

print("Something after the function.")

return wrapper

# Apply the decorator using the @ syntax

@my\_decorator

def say\_hello():

print("Hello!")

# Call the function

say\_hello()

# Output:

# Something before the function.

# Hello!

# Something after the function.

**Explanation:**

* my\_decorator() is a function that takes another function func() as an argument.
* Inside the my\_decorator(), we define an inner function wrapper() that adds functionality before and after calling the original func().
* The @my\_decorator syntax applies the decorator to the say\_hello() function. When say\_hello() is called, the code in the decorator is executed.

**Rewriting Without @ Syntax**

The above example with the @decorator syntax is equivalent to this:

python

Copy code

def say\_hello():

print("Hello!")

say\_hello = my\_decorator(say\_hello) # Manually applying the decorator

say\_hello() # Call the decorated function

**Decorators with Arguments**

If the function to be decorated takes arguments, the wrapper function must accept these arguments and pass them to the original function.

**Example: Decorator with Arguments**

python

Copy code

# Define a decorator

def my\_decorator(func):

def wrapper(\*args, \*\*kwargs):

print("Before the function.")

result = func(\*args, \*\*kwargs) # Call the original function with arguments

print("After the function.")

return result

return wrapper

@my\_decorator

def greet(name):

print(f"Hello, {name}!")

# Call the decorated function

greet("Alice")

# Output:

# Before the function.

# Hello, Alice!

# After the function.

**Explanation:**

* The wrapper() function accepts \*args and \*\*kwargs to handle any number of positional and keyword arguments.
* The original greet() function takes the argument name. The decorator now works for any function with arguments, since wrapper() passes \*args and \*\*kwargs to the original function.

**Practical Use Case: Timing Function Execution**

You can use decorators to measure the execution time of a function. This is a common use case for performance monitoring.

**Example: Timing a Function**

python

Copy code

import time

# Define a timing decorator

def timing\_decorator(func):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs) # Call the original function

end\_time = time.time()

print(f"Execution time: {end\_time - start\_time} seconds")

return result

return wrapper

@timing\_decorator

def slow\_function():

time.sleep(2) # Simulate a slow operation

print("Function complete")

# Call the decorated function

slow\_function()

# Output:

# Function complete

# Execution time: 2.00xxx seconds

**Explanation:**

* The timing\_decorator() calculates the execution time by recording the time before and after the function call.
* When slow\_function() is called, it sleeps for 2 seconds to simulate a slow task. The decorator prints how long it took to execute the function.

**Decorator with Parameters**

Sometimes, you may want to pass arguments to a decorator. In such cases, you need to create a function that returns a decorator.

**Example: Decorator with Parameters**

python

Copy code

# Define a decorator that accepts parameters

def repeat(n):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

for \_ in range(n): # Repeat n times

func(\*args, \*\*kwargs)

return wrapper

return decorator

@repeat(3) # Call the decorator with argument 3

def say\_hello():

print("Hello!")

# Call the decorated function

say\_hello()

# Output:

# Hello!

# Hello!

# Hello!

**Explanation:**

* The repeat(n) function is a **decorator factory** that takes an argument n. It returns the actual decorator that will repeat the decorated function n times.
* The say\_hello() function is decorated to repeat 3 times.

**Chaining Multiple Decorators**

You can apply multiple decorators to a single function by stacking them. The decorators are applied from top to bottom.

**Example: Chaining Decorators**

python

Copy code

def decorator\_one(func):

def wrapper(\*args, \*\*kwargs):

print("Decorator One")

return func(\*args, \*\*kwargs)

return wrapper

def decorator\_two(func):

def wrapper(\*args, \*\*kwargs):

print("Decorator Two")

return func(\*args, \*\*kwargs)

return wrapper

@decorator\_one

@decorator\_two

def greet(name):

print(f"Hello, {name}!")

# Call the decorated function

greet("Alice")

# Output:

# Decorator One

# Decorator Two

# Hello, Alice!

**Explanation:**

* @decorator\_one is applied first, then @decorator\_two. The output order follows the stacking order.

**Built-in Decorators in Python**

Python provides several built-in decorators, such as:

1. **@staticmethod**: Defines a static method.
2. **@classmethod**: Defines a class method.
3. **@property**: Used to define getters, setters, and deleters for class attributes.

**Example: Using @staticmethod**

python

Copy code

class MyClass:

@staticmethod

def static\_method():

print("This is a static method")

# Call the static method

MyClass.static\_method()

**Conclusion**

* **Decorators** are a powerful way to modify or extend the behavior of functions or methods in Python.
* They are commonly used for adding logging, enforcing access control, monitoring performance, and modifying behavior without changing the actual code of the decorated function.
* You can also pass arguments to decorators or chain multiple decorators for more complex behavior.

Decorators help improve the modularity and readability of your code.