Complete Guide to Decorators in Python

Table of Contents

1. Introduction

A decorator in Python is a function that takes another function and extends its behavior without explicitly modifying it. Decorators are used to implement cross-cutting concerns such as logging, validation, timing, and access control.

2. How Decorators Work

Decorators wrap a target function. Using the @decorator syntax is equivalent to assigning the function to the result of the decorator: f = decorator(f). When the decorated function is called, the wrapper runs, optionally performs actions before and after calling the original function, and returns a result.

```
Decorator flow (conceptual):

  @decorator
  def f(...):
      ...

is equivalent to:

  f = decorator(f)

Call flow: f() -> wrapper() -> original function
```

3. Simple Decorator Example

```
def my_decorator(func):
    def wrapper():
        print('Before function runs')
        func()
        print('After function runs')
    return wrapper

@my_decorator
def greet():
    print('Hello, World!')

greet()
```

Output: Before function runs Hello, World! After function runs

4. Decorator with Parameters

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print('Before function')
        result = func(*args, **kwargs)
        print('After function')
        return result
    return wrapper

@my_decorator
def greet(name):
    print(f'Hello, {name}!')

greet('Manivardhan')
```

5. Decorator with Return Value

If the original function returns a value, the wrapper must return it as well. If the wrapper does not return the result, the decorated function will return None.

```
def decorator(func):
    def wrapper(a, b):
        print('Before function')
        result = func(a, b)
        print('After function')
        return result
    return wrapper

@decorator
def add(a, b):
    return a + b

result = add(5, 3)
print('Result:', result)
```

6. Real-World Use Cases

Common uses of decorators include: - Logging: record function calls and arguments. - Validation: enforce argument types or constraints. - Timing: measure execution time for profiling. - Caching: memoize function results to avoid recomputation. - Access control: check permissions before executing a function.

7. Parameterized Decorators

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet():
    print('Hello!')

greet()
```

8. Chaining Multiple Decorators

```
def dec1(func):
    def wrapper(*args, **kwargs):
        print('dec1 before')
        result = func(*args, **kwargs)
        print('dec1 after')
        return result
    return wrapper

def dec2(func):
    def wrapper(*args, **kwargs):
        print('dec2 before')
        result = func(*args, **kwargs)
        print('dec2 after')
        return result
    return wrapper

@dec1
@dec2
def f():
    print('inside f')

f()
```

9. Summary Table

Concept	Description
Decorator	Wraps a function to modify its behavior
Wrapper	Inner function that adds logic
*args, **kwargs	Allow flexible arguments
return result	Ensure output is preserved
@syntax	Shortcut for applying decorators
Chaining	Apply multiple decorators
Parameterized Decorator	Accepts custom arguments

10. Key Takeaways

• Decorators make code cleaner and reduce repetition. • Always return the wrapped function's result if needed. • Use *args and **kwargs to support any signature. • Parameterized decorators allow customization. • Useful for logging, validation, timing, caching, and access control.