# 🚀 What is a Decorator?

At the deepest level:

**A decorator is a function that takes another function, adds extra behavior to it, and returns a new function — without modifying the original function's code.**

In simple words:

👉 **Decorator = function enhancer**

# 🔷 LEVEL 1 — Why Decorators Exist

Imagine you have many functions and you want to:

- log when they run
- measure execution time
- check permissions
- authenticate users

Without decorators, you would repeat code everywhere.

Example (bad approach):

```
def func1():
    print("Start")
    print("Function logic")
    print("End")

def func2():
    print("Start")
    print("Another logic")
    print("End")
```

Repeated code ❌

Python says:

"Functions are objects. Why not wrap them?"

That's where **decorators** come in.

# 🔷 LEVEL 2 — Functions are Objects (CRITICAL IDEA)

In Python:

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

This means:

- `greet` is a **function object**
- You can pass it as an argument
- You can return it from another function

Example:

```
def say_hello(func):
    func()

say_hello(greet)
```

This is the **foundation of decorators**.

# 🔷 LEVEL 3 — A Simple Decorator (Manual Way)

Let's build one slowly.

### Step 1: Write a wrapper

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

### Step 2: Apply it manually

```
def say_hi():
    print("Hi!")

say_hi = my_decorator(say_hi)
say_hi()
```

Output:

```
Before function
Hi!
After function
```

👉 We **did not change `say_hi()` code**, but its behavior changed.

That is a decorator.

# 🔷 LEVEL 4 — The `@decorator` Syntax (Python Magic)

Instead of:

```
say_hi = my_decorator(say_hi)
```

Python provides shortcut syntax:

```
@my_decorator
def say_hi():
    print("Hi!")
```

This means:

```
say_hi = my_decorator(say_hi)
```

Exactly the same thing.

# 🔷 LEVEL 5 — How Decorators Work Internally

When Python sees:

```
@my_decorator
def say_hi():
    print("Hi!")
```

It does this:

1. Creates `say_hi` function
2. Passes it to `my_decorator`
3. Stores returned `wrapper` as `say_hi`

So now:

```
say_hi()   # actually calls wrapper()
```

Original function is **wrapped**.

# 🔷 LEVEL 6 — Decorators with Arguments

Most real functions have parameters.

So wrapper must accept them.

### Correct way:

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

Example:

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

print(add(2, 3))
```

Output:

```
Before
After
5
```

# 🔷 LEVEL 7 — Decorator with Return Value

Important rule:

Wrapper should return the result of the original function.

Otherwise:

- decorated function returns `None`

Example mistake ❌:

```
def wrapper():
    func()   # no return
```

Correct ✔:

```
return func()
```

# 🔷 LEVEL 8 — Real-World Decorator Example (Timing)

```
import time

def time_it(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Execution time: {end - start}")
        return result
    return wrapper

@time_it
def slow_func():
    time.sleep(2)

slow_func()
```

This is used **everywhere in production code**.

# 🔷 LEVEL 9 — Decorator with Arguments (Advanced but important)

Example:

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

Usage:

```
@repeat(3)
def hello():
    print("Hello")

hello()
```

Output:

```
Hello
Hello
Hello
```

Structure:

```
repeat(n) → decorator → wrapper
```

# 🔷 LEVEL 10 — Important Problem: Function Metadata Loss

After decoration:

```
print(say_hi.__name__)
```

Output:

```
wrapper
```

Original name lost ❌

### Solution: `functools.wraps`

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

Now metadata is preserved.

# 🔷 LEVEL 11 — Built-in Decorators in Python

You already use decorators without realizing it:

| Decorator | Purpose |
|---|---|
| `@staticmethod` | No `self` |
| `@classmethod` | Uses `cls` |
| `@property` | Access method like attribute |

Example:

```
class Person:
    @property
    def name(self):
        return "Vrund"
```

# 🔥 Final Cheat Sheet

| Concept | Meaning |
|---|---|
| Decorator | Function that modifies another function |
| Wrapper | Inner function that adds behavior |
| `@` | Syntax sugar |
| `*args, **kwargs` | Support any arguments |
| `functools.wraps` | Preserve metadata |

# One-line Core Meaning

**A decorator is a Python feature that allows you to extend a function's behavior without changing its code, by wrapping it inside another function.**