🧠 Lecture 12: Decorators in Python

🔰 Introduction

A **decorator** is a special function that **modifies or enhances** another function's behavior — without changing its original code.

🎯 Think of it like putting a gift 🎁 in a fancy wrapper 🎀 — same gift, better presentation!

🌟 1. Why Use Decorators?

✅ **Code Reusability** – Apply same enhancement to many functions
✅ **Code Readability** – Keeps core logic and extra functionality separate
✅ **Real Use Cases** – Logging, authentication, timing, etc.

□ 2. Core Concepts

✔ Function = First-Class Citizen

You can:

- Assign functions to variables
- Pass them as arguments
- Return them from other functions

□ 3. Basic Decorator Example

```
python
CopyEdit
def greet():
    return 'Hello!'

def call_function(func):
    return func()

print(call_function(greet))  # → Hello!
```

➡ We passed the function `greet` as an argument!

🎁 Decorator Structure

```
python
CopyEdit
def decorator_function(original_function):
    def wrapper():
        print(f"Logging: Function '{original_function.__name__}' is being
called.")
        return original_function()
    return wrapper

@decorator_function
def say_hello():
    print("Hello, John!")

say_hello()
```

📤 Output:

```
pgsql
CopyEdit
Logging: Function 'say_hello' is being called.
Hello, John!
```

□ 4. Preserve Metadata

To keep the original function's name, docstring etc., use:

```
python
CopyEdit
from functools import wraps

def decorator_function(func):
    @wraps(func)
    def wrapper():
        # Do something extra
        return func()
    return wrapper
```

🕒 5. Real-Life Use Case: Timing Function

```
python
CopyEdit
import time
```

```
def timer_decorator(func):
    def wrapper():
        start = time.time()
        func()
        end = time.time()
        print("Execution Time:", end - start)
    return wrapper

@timer_decorator
def slow_function():
    time.sleep(2)
    print("Done!")

slow_function()
```

🔁 6. Decorator with Arguments

✅ Example: Repeat a Function

```
python
CopyEdit
def repeat(times):
    def decorator(func):
        def wrapper():
            for _ in range(times):
                func()
        return wrapper
    return decorator

@repeat(3)
def greet():
    print("Hello!")
```

📤 Output:

```
CopyEdit
Hello!
Hello!
Hello!
```

💡 Best Practices

Do ✅	Avoid ❌
Use `@wraps` to keep metadata	Too many nested decorators

Do ✅	Avoid ❌
Reuse decorators wisely	Making them too complex
Keep logic clean & modular	Mixing core logic and enhancement

□ Final Thoughts

✅ **Decorators** = power tool for writing **cleaner**, **modular**, and **maintainable** code.
💡 Use them when multiple functions need similar *extra* behavior.