📆 DAY 03 — Decorators (Function Wrappers)

Goal : Understand how Python *wraps functions* and how to use decorators safely

Why ML / Software Engineers need this:

- Logging
- Timing / profiling
- Authentication
- Validation
- Caching
- API frameworks (FastAPI, Flask) use decorators everywhere

1️⃣ What Is a Decorator

A decorator is:

A function that takes another function and extends its behavior without modifying it.

simple idea:

```
@decorator
def my_function():
    ...
```

Is the same as :

```
my_function = decorator(my_function)
```

That’s it. No magic.

2️⃣ Functions Are Objects (Critical Concept)

In Python:

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

x = greet
x()

#  output
Hello
```

Functions can be:

- Passed as arguments
- Returned from other functions
- Stored in variables

Decorators depend on this.

3️⃣ First: A Simple Wrapper (No @ Yet)

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

Usage:

```
def say_hello():
    print("Hello!")

say_hello = my_decorator(say_hello)
say_hello()

# output
Before function runs
```

```
Hello!
After function runs
```

4️⃣ Using the `@` Syntax (Cleaner)

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

@mydecorator
def say_hello():
    print("Hello!")
```

Same results, much cleaner.

5️⃣ Decorators with Arguments

Most real function take arguments.

❌ This will break:

```
def wrapper():
    func()
```

✅ Correct way:

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

`*args, **kwargs` = works for ANY function signature

This is **mandatory in production code.**

6️⃣ Real Engineer Example 1: Timing Decorator

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f}s")
        return result
    return wrapper
```

Usage:

```
@timer
def slow_function():
    time.sleep(1)

slow_function()

# output
slow_function took 1.000x s
```

🧠 ML Insight

This is used to :

- Time training loops
- Benchmark models
- Profile data loading

7️⃣ Real Engineer Example 2: Logging Decorator

```
def log_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
```

```
        return func(*args, **kwargs)
    return wrapper
```

Usage:

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

8️⃣ Preserving Function Metadata (`functools.wraps`)

Without this:

```
print(add.__name__)

# Output:
wrapper
```

❌ Bad for debugging, docs, APIs.

✅ Fix:

```
from functools import wraps

def log_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
```

Alwaus use `@wraps` in real projects.

9️⃣ Decorators in the Real World

You’ve already seen them:

```
@app.get("/predict")
defpredict():
```

```
    ...
```

```
@lru_cache
defexpensive_function():
    ...
```

```
@staticmethod
@classmethod
```

Frameworks = decorators everywhere.