🧵 Python Decorators: Full Notes with Code & Emojis ✨

📘 What is a Decorator?

🔧 A **decorator** in Python is a **function that modifies the behavior of another function or method** without changing its actual code.

Think of decorators as **wrappers** 🎁 you put around your functions to give them superpowers!

🧠 Why Use Decorators?

- ✅ Code Reusability
- ✅ Cleaner and More Readable Code
- ✅ Separation of Concerns
- ✅ DRY Principle (Don’t Repeat Yourself)

⚙️ Basic Syntax

```python
def decorator_function(original_function):
    def wrapper_function():
        print("Before the original function")
        original_function()
        print("After the original function")
    return wrapper_function

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

say_hello()
```

🧾 Output:

```
Before the original function
Hello!
After the original function
```

🧰 Core Concepts

Concept	Emoji	Description
Functions are first-class objects	🧱	Functions can be passed as arguments
Nested functions	🥪	Functions inside functions
Returning functions	🎯	Functions can return other functions
Decorators	🎁	Wrap a function to extend its behavior

🎯 Real-World Use Cases

Use Case	Emoji	Description
Timing	⏱️	Measure how long a function takes
Debugging	🐞	Log function calls and arguments
Caching	🧠	Avoid recomputation by storing results
Authentication	🔐	Restrict access to sensitive routes (e.g., in Flask)
Validation	✅	Check input before running logic

📑 Decorator Examples with Code and Explanation

---

1. ⏱️ Time Measuring Decorator

```python
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"⏱️ Execution time: {end - start:.4f} seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(2)
    return "Done!"

print(slow_function())
```

🔍 **Why?** Useful for performance optimization.

---

2. 🐞 Debugging Decorator

```python
def debug(func):
    def wrapper(*args, **kwargs):
        print(f"🐞 Calling {func.__name__} with args={args} kwargs={kwargs}")
        return func(*args, **kwargs)
    return wrapper

@debug
def greet(name, msg="Hello"):
    return f"{msg}, {name}!"

print(greet("Darshan", msg="Hi"))
```

🔍 **Why?** Helps trace bugs and understand flow.

---

3. 🧠 Cache (Memoization) Decorator

```python
def cache(func):
    memory = {}
    def wrapper(*args):
        if args in memory:
            print("🧠 Returning cached result")
            return memory[args]
        print("🔁 Computing new result")
        result = func(*args)
        memory[args] = result
        return result
    return wrapper

@cache
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

🔍 **Why?** Boosts performance of recursive functions.

✴️ Advanced Decorator Features

🔁 Decorators with Arguments

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

@repeat(3)
def say_hi():
    print("Hi!")

say_hi()
```

🪆 **Nested decorators**: Allows parameterized behavior.

📌 Preserving Metadata with `functools.wraps`

```
from functools import wraps

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

@debug
def my_func():
    """This is a docstring"""
    pass

print(my_func.__name__)  # ✅ my_func
print(my_func.__doc__)   # ✅ This is a docstring
```

⚠️ **Without `@wraps`**, you lose metadata like function name and docstring!

🔐 Decorators in Flask (Web Example)

```
from flask import Flask, request
app = Flask(__name__)

def require_api_key(func):
    def wrapper(*args, **kwargs):
        if request.args.get('api_key') != 'secret123':
            return "Unauthorized", 403
        return func(*args, **kwargs)
    return wrapper

@app.route('/data')
```

```python
@require_api_key
def get_data():
    return {"data": "Here it is!"}
```

🧱 **Why?** Used for security and access control.

🧩 Multiple Decorators on One Function

```python
@timer
@debug
def example():
    print("Running example...")

example()
```

🎯 **Order Matters!** Outer decorator runs first.

🧪 Create Your Own Utility Decorators

Want to build:

- 🔁 Retry decorator?
- 💤 Delay execution decorator?
- 📈 Log-to-file decorator?

Let me know and I'll help you build it!

📌 Summary Table

Feature	Emoji	Purpose
`@timer`	⏱️	Measure execution time
`@debug`	🐞	Log calls and arguments
`@cache`	🧠	Store and reuse return values
`@wraps` from functools	📌	Preserve original function info
Parametrized decorator	🔁	Customize decorator behavior

🚀 You're Now a Decorator Wizard! 🧙‍♂️🌀

Keep practicing and try creating decorators for:

- Authentication
- Rate Limiting

- Retry Logic
- Logging to Files
- Access Control

🧙 Master Python Decorators with Magical Mini-Quests 🪄

Decorators in Python are like magical cloaks 🧥 that wrap your functions to give them superpowers — without changing their core logic. Ready to enchant your functions?

▶ Details

1. ⏱️ Time Tracker Spell: Measure Execution Time

🧩 Problem:

Write a decorator `@timer` that measures how long a function takes to run and prints the duration in seconds.

✅ Use Case:

Perfect for **benchmarking** heavy or slow operations like data processing, simulations, or API requests.

🧠 Explanation:

- Use `time.time()` to capture start and end times.
- Calculate the difference.
- Print the result before returning the original function's output.

🧪 Code:

```python
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"⏱️ Function '{func.__name__}' took {end_time - start_time:.4f} seconds to run")
        return result
    return wrapper

@timer
def long_task():
    time.sleep(2)
    return "🛠️ Task completed!"

print(long_task())
```

▶ Details

2. 🐞 Debug Mirror Charm: Trace Function Calls

🧩 Problem:

Create a decorator `@debug` that prints the function name, arguments, and keyword arguments every time the function is called.

✅ Use Case:

Debugging mysterious bugs in functions or APIs. This is your real-time x-ray vision. 🔍

🧠 Explanation:

- Access `*args` and `**kwargs` to log input values.
- Print before running the function.

🧪 Code:

```python
def debug(func):
    def wrapper(*args, **kwargs):
        args_list = ', '.join([repr(a) for a in args])
        kwargs_list = ', '.join([f"{k}={v!r}" for k, v in kwargs.items()])
        all_args = ', '.join(filter(None, [args_list, kwargs_list]))
        print(f"🐞 Calling {func.__name__}({all_args})")
        return func(*args, **kwargs)
    return wrapper

@debug
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

print(greet("Darshan", greeting="Hi"))
```

▶ Details

3. 🧠 Memory Potion: Cache Function Results

🧩 Problem:

Create a decorator `@cache` that stores the result of function calls with given arguments, and **returns the cached result** if the function is called again with the same inputs.

✅ Use Case:

Highly useful for **recursive functions**, **data fetching**, and any **CPU-heavy** tasks with repeated inputs.

🧠 Explanation:

- Use a dictionary to store `(args, kwargs)` as the key and the result as the value.
- Return from cache if key exists; otherwise compute and store.

🧪 Code:

```python
def cache(func):
    memory = {}
    def wrapper(*args):
        if args in memory:
            print(f"🧠 Using cached result for {func.__name__}{args}")
            return memory[args]
        print(f"🧪 Calculating new result for {func.__name__}{args}")
        result = func(*args)
        memory[args] = result
        return result
    return wrapper

@cache
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(f"Result: {fibonacci(10)}")
```

📑 Summary

🪄 Decorator	Purpose	Real-World Use
`@timer`	Track how long a function runs	Benchmarking, performance testing
`@debug`	Log function calls and parameters	Debugging, tracing
`@cache`	Store and reuse function results	Recursive ops, optimization

---

📘 Python Decorators Practice Workbook + Advanced Patterns Guide

> **Level:** Beginner ➡️ Intermediate ➡️ Pro **Includes:** Practice problems 🧩, advanced use cases ⚡, and reusable patterns 🔁

🎯 Section 1: Core Practice – Build Your Decorator Skills

✅ 1. Log Function Calls

Problem: Write a `@logger` decorator that logs:

- Function name
- Arguments
- Return value

Hint: Use `print()` and `repr()`.

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

add(3, 7)
```

✅ 2. Authorize Access

Problem: Write a decorator `@admin_only` that allows function access **only if** a global `user` has role `"admin"`.

```
user = {"name": "Darshan", "role": "admin"}

@admin_only
def delete_database():
    return "💧 DB deleted!"
```

✅ 3. Delay Function Execution

Problem: Write a decorator `@delay(seconds)` that delays function execution by given seconds using `time.sleep()`.

```
@delay(2)
def say_hi():
    print("Hi after 2 seconds")
```

✅ 4. Retry on Failure

Problem: Build a decorator `@retry(n)` that retries a function `n` times if it raises an exception.

```
@retry(3)
def fragile():
    if random.random() < 0.7:
        raise ValueError("❌ Random failure")
    return "✅ Success"
```

⚡ Section 2: Advanced Decorator Patterns

🔁 5. Decorator with Parameters

Problem: Create a `@tag(msg)` decorator that prints a custom message before executing the function.

```
@tag("🔥 CRITICAL TASK STARTED")
def run_backup():
    print("Running backup...")
```

🧠 6. Cache with Expiry

Problem: Build a `@cache_with_expiry(seconds)` that caches return values for a limited time (like LRU cache with TTL).

```
@cache_with_expiry(5)
def fetch_data():
    print("Fetching fresh data...")
    return "Data at " + str(time.time())
```

🔐 7. Function Access Control (Auth)

Problem: Write a `@require_login` decorator that prevents access unless a `user["logged_in"]` is `True`.

```
user = {"logged_in": False}

@require_login
def view_profile():
    return "User profile data"
```

🧾 8. Decorator for Logging to File

Problem: Create `@log_to_file("logs.txt")` that writes logs to a file instead of printing them.

```
@log_to_file("logs.txt")
def signup(username):
    return f"{username} signed up!"
```

🧪 9. Decorate Class Methods

Problem: Use decorators to log all methods of a class. Bonus: Create a class decorator `@log_all_methods`.

```python
@log_all_methods
class BankAccount:
    def deposit(self, amt): ...
    def withdraw(self, amt): ...
```

✴ 10. Nesting Decorators

Problem: Create two decorators `@upper_case` and `@add_greeting` and apply both.

```python
@add_greeting
@upper_case
def get_name():
    return "darshan"
# Output: "Hello, DARSHAN"
```

📦 Bonus Patterns

Pattern	Use Case	Emoji
`@singleton`	Ensure only one instance	🧍
`@throttle(seconds)`	Limit how often function runs	🚦
`@validate_types`	Enforce function argument types	🔍
`@profile`	Print memory/cpu stats of a function	🧮
`@inject_config`	Inject global settings/configs	🔌

🧵 Decorators in **FastAPI**

> Supercharge your APIs with reusable logic like auth, logging, rate limiting, and more!

🚀 Why Use Decorators in FastAPI?

FastAPI is built on **Python functions** (path operations), so decorators are the perfect way to:

- 🔐 Authenticate users
- 🐞 Log requests
- 🚦 Apply rate-limiting
- 🔁 Retry failing operations
- 🎯 Apply validation or custom checks

✅ Example 1: Logging Requests Decorator

```python
from fastapi import FastAPI, Request
import time

app = FastAPI()

def log_request(func):
    async def wrapper(*args, **kwargs):
        start = time.time()
        result = await func(*args, **kwargs)
        duration = time.time() - start
        print(f"📨 Request to {func.__name__} took {duration:.2f}s")
        return result
    return wrapper

@app.get("/hello")
@log_request
async def hello():
    return {"msg": "Hello, Darshan!"}
```

🧠 Works great for tracing performance and debugging!

🔐 Example 2: Auth Decorator

```python
from fastapi import Request, HTTPException

def require_api_key(func):
    async def wrapper(request: Request, *args, **kwargs):
        api_key = request.headers.get("x-api-key")
        if api_key != "supersecret":
            raise HTTPException(status_code=403, detail="🔐 Forbidden: Invalid API Key")
        return await func(request, *args, **kwargs)
    return wrapper

@app.get("/secure")
@require_api_key
async def secure_endpoint(request: Request):
    return {"msg": "Welcome to the vault 🔐"}
```

🔐 Add to any route for simple header-based access control.

🚦 Example 3: Simple Rate Limiter

```
import time
user_last_call = {}

def rate_limit(seconds: int):
    def decorator(func):
        async def wrapper(request: Request, *args, **kwargs):
            ip = request.client.host
            now = time.time()
            if ip in user_last_call and now - user_last_call[ip] < seconds:
                raise HTTPException(status_code=429, detail="⏱️ Too Many
Requests")
            user_last_call[ip] = now
            return await func(request, *args, **kwargs)
        return wrapper
    return decorator

@app.get("/limited")
@rate_limit(5)
async def limited(request: Request):
    return {"msg": "This route is rate-limited by IP"}
```

⚙️ Reusable, IP-based throttling!

📌 Best Practices for FastAPI Decorators

Tip	Explanation
Use `async def` in wrappers	To support FastAPI's async stack
Add `functools.wraps(func)`	To preserve function metadata
Decorate outside FastAPI classes	Keeps code modular and clean
Avoid blocking code	Use `asyncio.sleep()` if needed

🪄 Function Factories & Dynamic Decorators

> Build decorators **on-the-fly** with **custom parameters** or **logic generators**.

🔁 What's a Function Factory?

A **function factory** returns a function based on parameters. Think of it like a factory that builds decorators dynamically!

🧪 Example 1: `@tag(msg)` — Custom Logging Tag

```python
def tag(msg):
    def decorator(func):
        def wrapper(*args, **kwargs):
            print(f"[{msg}] 🚀 Calling {func.__name__}")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@tag("CRITICAL")
def run_job():
    print("⚙️ Running job...")

run_job()
```

💡 Output:

```
[CRITICAL] 🚀 Calling run_job
⚙️ Running job...
```

🔁 Example 2: Retry Logic with Max Tries

```python
import random

def retry(max_tries):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for i in range(max_tries):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"❌ Attempt {i+1} failed: {e}")
            raise Exception("🛑 All attempts failed")
        return wrapper
    return decorator

@retry(3)
def unstable():
    if random.random() < 0.7:
        raise ValueError("Random fail")
    return "✅ Success"
```

```
print(unstable())
```

🔥 Use Cases of Dynamic Decorators

Decorator Type	Factory Example
`@retry(n)`	Control error tolerance
`@timeout(seconds)`	Auto-cancel slow tasks
`@require_role("admin")`	Role-based access
`@log_to_file(path)`	Dynamic log paths
`@cache_with_expiry(seconds)`	Smart memoization

✨ Tips for Function Factories

- Always **nest the wrapper** inside the parameterized outer function.
- Preserve metadata using `functools.wraps()`.
- Return the **inner decorated function**.

📦 Bonus Utilities with `functools`

```
from functools import wraps

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

Why? ✅ Preserves:

- `__name__`
- `__doc__`
- `__annotations__`
