**Decorators in Python**

Decorators in Python are a powerful tool that allows the modification of functions or methods without changing their actual code. They are used to extend or modify the behavior of functions dynamically.

**1. What is a Decorator?**

A **decorator** is a function that takes another function as an argument and extends or alters its behavior without explicitly modifying it.

Decorators are commonly used in:

* Logging
* Access control & authentication
* Memoization (caching)
* Measuring execution time
* Debugging

**2. Basic Structure of a Decorator**

A decorator is a function that takes another function as an argument and returns a new function.

**Example of a simple decorator:**

def my\_decorator(func):

def wrapper():

print("Something before the function runs.")

func()

print("Something after the function runs.")

return wrapper

@my\_decorator # Applying the decorator

def say\_hello():

print("Hello!")

say\_hello()

**Output:**

Something before the function runs.

Hello!

Something after the function runs.

**3. Using @ Syntax**

Instead of manually wrapping a function, we use the @decorator\_name syntax:

@my\_decorator

def say\_hello():

print("Hello!")

This is equivalent to:

say\_hello = my\_decorator(say\_hello)

**4. Decorators with Arguments**

If a function has arguments, the wrapper function should accept \*args and \*\*kwargs.

def smart\_divide(func):

def wrapper(a, b):

if b == 0:

print("Cannot divide by zero!")

return

return func(a, b)

return wrapper

@smart\_divide

def divide(a, b):

return a / b

print(divide(10, 2)) # 5.0

print(divide(10, 0)) # Cannot divide by zero!

**5. Multiple Decorators (Chaining Decorators)**

You can apply multiple decorators to a function by stacking @decorator1, @decorator2, etc.

def decorator1(func):

def wrapper():

print("Decorator 1")

func()

return wrapper

def decorator2(func):

def wrapper():

print("Decorator 2")

func()

return wrapper

@decorator1

@decorator2

def hello():

print("Hello!")

hello()

**Output:**

Decorator 1

Decorator 2

Hello!

**6. Function Decorators vs Class Decorators**

**Function-based Decorators**

Function decorators are applied to functions and modify their behavior.

**Class-based Decorators**

Instead of defining a function, a decorator can be a class with a \_\_call\_\_ method.

class MyDecorator:

def \_\_init\_\_(self, func):

self.func = func

def \_\_call\_\_(self, \*args, \*\*kwargs):

print("Before function call")

result = self.func(\*args, \*\*kwargs)

print("After function call")

return result

@MyDecorator

def greet(name):

print(f"Hello, {name}!")

greet("Alice")

**Output:**

Before function call

Hello, Alice!

After function call

**7. Built-in Decorators**

Python provides several built-in decorators:

* @staticmethod → Defines a static method in a class.
* @classmethod → Defines a class method.
* @property → Defines a getter method.

Example:

class Person:

def \_\_init\_\_(self, name):

self.\_name = name

@property

def name(self):

return self.\_name

p = Person("Alice")

print(p.name) # Alice

**8. Real-world Example: Logging Decorator**

import time

def log\_execution\_time(func):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

print(f"Execution time: {end\_time - start\_time:.4f} seconds")

return result

return wrapper

@log\_execution\_time

def compute\_square(n):

return [x\*\*2 for x in range(n)]

compute\_square(1000000)

**9. functools.wraps – Preserving Function Metadata**

When using decorators, function metadata (\_\_name\_\_, \_\_doc\_\_, etc.) can be lost. To prevent this, use functools.wraps.

from functools import wraps

def my\_decorator(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

"""Wrapper function"""

print("Before function")

return func(\*args, \*\*kwargs)

return wrapper

@my\_decorator

def example():

"""Original function"""

print("Hello!")

print(example.\_\_name\_\_) # example

print(example.\_\_doc\_\_) # Original function

**10. Summary**

* **Decorators** modify functions dynamically.
* Use @decorator\_name to apply decorators.
* Use \*args and \*\*kwargs for flexibility.
* Stack multiple decorators for layered functionality.
* Use functools.wraps to preserve function metadata.