Python Decorators 📌

Introduction to Decorators 📌

A **decorator** in Python is a function that wraps another function to **modify its behavior** without changing its code. It is used to enhance functions dynamically.

Why Use Decorators? 📌

✔ Code reusability
✔ Keep functions clean
✔ Add pre/post-processing logic
✔ Useful in logging, authentication, performance timing, etc.

Basic Syntax of a Decorator

A decorator takes a function as input, defines a **wrapper function**, and returns it.

```
def decorator_name(func):
    def wrapper():
        # Code before function execution
        result = func()
        # Code after function execution
        return result
    return wrapper
```

Example 1: Logging Execution

```python
def logtime(func):
    def wrapper():
        print("Before function call")
        val = func()
        print("After function call")
        return val
    return wrapper

@logtime  # Applying decorator
def hello():
    print("Hello!")

hello()
```

Output:

```
Before function call
Hello!
After function call
```

Example 2: Measuring Execution Time

```python
import time

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

@timer
def slow_function():
    time.sleep(2)
    print("Function completed!")

slow_function()
```

Output:

```
Function completed!
Execution time: 2.0001 seconds
```

Passing Arguments to Decorated Function

For functions that take arguments, use `*args` and `**kwargs`.

```python
def log_args(func):
    def wrapper(*args, **kwargs):
        print(f"Called with args: {args}, kwargs: {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@log_args
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet("Alice", 30)
```

Output:

```
Called with args: ('Alice', 30), kwargs: {}
Hello Alice, you are 30 years old.
```

Passing Arguments to Decorator

To pass arguments **to the decorator itself**, wrap it in another function.

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

@repeat(3)  # Repeat function 3 times
def say_hi():
    print("Hi!")

say_hi()
```

Output:

```
Hi!
Hi!
Hi!
```

Using Multiple Decorators (Stacking)

You can apply multiple decorators to a function.

```python
def bold(func):
    def wrapper():
        return f"<b>{func()}</b>"
    return wrapper

def italics(func):
    def wrapper():
        return f"<i>{func()}</i>"
    return wrapper

@bold
@italics
def text():
    return "Hello"

print(text())
```

Output:

Hello

Class-Based Decorators

Instead of functions, decorators can also be implemented using **classes**.

```python
class Logger:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f"Executing {self.func.__name__}")
        return self.func(*args, **kwargs)

@Logger
def say_hello():
    print("Hello, World!")
```

```
say_hello()
```

Output:

```
Executing say_hello
Hello, World!
```

Built-in Decorators in Python

Python provides several built-in decorators:

1. **@staticmethod** – Defines a method that doesn't modify class state.
2. **@classmethod** – Allows access to class variables.
3. **@property** – Defines a getter method for an attribute.

Example of `@property`

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

p = Person("Alice")
print(p.name)  # Alice
```

Key Takeaways

✔ **Decorators wrap functions** and modify behavior without changing them.
✔ **Useful for logging, authentication, and timing.**
✔ **Use `*args` and `**kwargs`** to support flexible function arguments.
✔ **Multiple decorators can be stacked.**
✔ **Class-based decorators** allow for stateful decorators.