Day 35

Decorators in Python:

Decorators in Python — From Basics

A decorator in Python is a function that modifies the behaviour of another function or method. Think of it as "wrapping" a function to add extra functionality without changing its original code.

Functions are first-class objects: In Python, functions are objects, which means:

- You can assign a function to a variable.
- You can pass a function as an argument to another function.
- You can return a function from another function.

Example:

```
def greet(name):
    return f"Hello, {name}!"
# Assigning function to a variable
say_hello = greet
print(say_hello("Alice"))  # Output: Hello, Alice!
```

Functions inside functions: You can define a function inside another function:

Example:

```
def outer():
    def inner():
        return "I'm inside!"
    return inner

my_func = outer()
print(my_func())  # Output: I'm inside!
```

Notice outer() returns a function inner without calling it (no parentheses after inner when returning).

Passing functions as arguments: You can pass a function to another function:

```
def greet(name):
    return f"Hello, {name}!"
def call_func(func, name):
    return func(name)
print(call_func(greet, "Bob"))  # Output: Hello, Bob!
```

What is a decorator?

A decorator is a function that takes another function and extends its behavior without explicitly modifying it. Think of it like wrapping a gift: the gift is the original function, and the wrapping is the decorator.

Example: decorating manually.

```
def decorator(func):
    def wrapper():
        print("Before calling the function")
        func()  # Call the original function
        print("After calling the function")
    return wrapper
def say_hello():
    print("Hello!")
# Decorating manually
decorated_function = decorator(say_hello)
decorated_function()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS E:\Python\Day31-40\Day35> python main.py
Before calling the function
Hello!
After calling the function
```

This code shows how a decorator works in Python. The decorator function takes another function (func) as input and wraps it inside a new function called wrapper. The wrapper adds extra behavior by printing a message before and after calling the original function. When say_hello is passed to decorator, it becomes decorated_function. So when decorated_function() is called, it first prints “Before calling the function”, then runs say_hello() (which prints “Hello!”), and finally prints “After calling the function”.

Example: Using the @ syntax. Python provides a shorthand for decorators using the @ symbol:

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        result = func(*args, **kwargs)
        print("After function call")
        return result
    return wrapper
@decorator
def greet(name):
    print(f"Hello, {name}!")
greet("Alice")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS E:\Python\Day31-40\Day35> python main.py
Before function call
Hello, Alice!
After function call
```

This code uses a decorator to add extra behavior to a function. The decorator takes a function and wraps it inside wrapper, which can accept any arguments using *args and **kwargs. When greet("Alice") is called, it actually runs the wrapper function: it prints “Before function call”, then calls the original greet function with the name "Alice" (which prints “Hello, Alice!”), and finally prints “After function call”. The @decorator line is just a shortcut for applying the decorator to greet.

Summary:

1. Functions are first-class objects.
2. You can define functions inside functions.
3. Functions can be passed as arguments or returned from other functions.
4. A decorator wraps a function to modify/extend its behavior.
5. Use @decorator_name to apply a decorator in Python.
6. Use *args and **kwargs in the wrapper to handle functions with parameters.

--The End--