Introduction

A decorator is a function that modifies or enhances the behavior of another function without modifying its source code.
In simple words Decorators are a powerful feature in Python that allows us to extend or modify the behavior of functions or methods without changing their actual code. They help in code reusability, readability, and can be applied to various tasks like logging, authentication, or timing functions.

Key Features of Decorators:

- Code Reusability: Decorators allow you to reuse the same function logic in multiple places without repeating the code.
- Code Readability: They make the code cleaner by separating the core logic of a function and its enhancement logic (e.g., logging, authentication).
- Use Cases: Can be used for logging, authentication, function timing, and more.

Core Concepts:

Function as First-Class Citizen:

- Functions can be used as variables, passed as arguments to other functions, and returned from functions.

Decorator Syntax:

1. Define the Decorator Function:
    - Assigned to Variables: Functions can be assigned to variables like any other object
    - Passed as Arguments: Functions can be passed as arguments to other functions
    - Returned from Functions: Functions can be returned from other functions
2. How It Works:
    - When a function is passed to a decorator, its behavior is extended without modifying the original function.

Example of Decorator:

```
def greet(): return 'Hello!'
def call_function(func):
    return func()
print(call_function(greet))
```

Output:

- Logging: Function 'greet' is being called.
- Hello, John!

Key Points in Using Decorators:

- Preserving Function Metadata:
    - Use functools.wraps() to preserve the metadata of the original function when decorators are applied.
- Avoid Unnecessary Nesting of Decorators:
    - Multiple decorators can make the code hard to follow. Avoid excessive nesting.

Practical Use Cases:

Logging and Time Measurement:

- Logging Function Call:

```
def decorator_function(original_function):
    def wrapper():
        # Add extra functionality
        return original_function()
    return wrapper
```

- Time Measurement:

```
import time
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                func(*args, **kwargs)
        return wrapper
    return decorator
```

Passing Arguments to Decorators

- Extra Function for Argument Passing:
    - A decorator can take arguments, for example, to repeat a function multiple times:

```
@repeat(3)
def greet():
    print('Hello!')
```

- Output:
    - Hello
    - Hello
    - Hello

Best Practices for Decorators:

- Use decorators to extend functionality without modifying the original code.
- Preserve function metadata using `functools.wraps()`.
- Avoid unnecessary nesting of decorators to keep the code readable and maintainable.

Conclusion:

- Decorators provide a clean, reusable way to add additional functionality to functions, such as logging or timing, without changing the core function.
- They are an essential concept for writing cleaner, more modular code in Python, especially when the same functionality needs to be applied to multiple functions.