PYTHON DECORATORS

Understanding and Implementing Decorators in Python

MUKESH KUMAR

Agenda

- ✓ Introduction to Decorators
- ✓ Why Use Decorators?
- ✓ How Decorators Work
- ✓ Chaining Multiple Decorators
- ✓ Decorators with Arguments
- ✓ Types of Decorators
- ✓ Different Ways to Apply Decorators
- ✓ Practical Use Cases of Decorators
- ✓ Built-in Decorators in Python

What are Decorators?

- Decorators in Python are a powerful and elegant feature that allow modification or extension of the behavior of functions or methods without changing their actual code.

What are Decorators?

- A decorator is a function that takes another function as an argument, modifies or extends its behavior, and returns the modified function.

- In simple terms, it wraps a function with additional behavior, without modifying the original function’s code.

Syntax

- The @ symbol (syntactic sugar) is used to apply a decorator to a function.

```
@decorator_name
def my_function():
    pass
```

- This is equivalent to:

```
my_function = decorator_name(my_function)
```

Why Use Decorators?

- **Separation of Concerns**: Keep core functionality clean and reusable.
- **Code Reusability**: Apply common behavior (e.g., logging, validation, timing) across multiple functions without repetition.
- **Enhance Readability**: Keep code clean by abstracting repetitive logic.

Higher-Order Functions

What are Higher-Order Functions?

- Decorators are higher-order functions, meaning they take another function as input and return a new function with updated behavior.
- A higher-order function can accept a function as an argument or return a function as a result (or both).
- Example:

```python
def decorator(func):
    def wrapper():
        print("Before the function is called")
        func()
        print("After the function is called")
    return wrapper
```

How Decorators Work

Step-by-step Process:

- A function is passed to another function (decorator).
- The decorator modifies or extends the behavior of the original function.
- The original function is called within the decorator, with added behavior around it.

Simple Decorator Example

- Refer Notebook simple_Decorator_example.ipynb

Chaining Multiple Decorators

- Refer notebook Chaining_multiple_decorators.ipynb

Decorators with Arguments

- Refer notebook decoratorswithargument.ipynb

Types of Decorators

- Function Decorators
- Class Method Decorators
- Property Decorators
- Method Decorators

Types of Decorators

Function Decorators

- Modify the behavior of functions.
- Example: Logging, timing, validation.

Class Method Decorators

- Used with class methods to modify the behavior of the method.
- Common Built-In Decorators: @staticmethod, @classmethod.

Types of Decorators

Property Decorators

- Used to define properties in classes, allowing a method to be accessed like an attribute.
- Common Built-In Decorator: @property.

Method Decorators

- These apply to methods of a class, modifying their behavior.
- Example: Adding additional behavior to instance methods.

Different Ways to Apply Decorators

- Refer notebook Diff_way_to_apply_decorators.ipynb

Practical Use Cases of Decorators

- Refer notebook Decorators_Applications.ipynb

Built-in Decorators in Python

- Examples of Built-In Decorators:
- - @staticmethod: Used to define a static method in a class.
- - @classmethod: Used to define a class method.
- - @property: Converts a method into a read-only property.

Summary

- Decorators allow you to modify or extend the behavior of functions and methods in a clean and maintainable way.
- They help promote code reusability, separation of concerns, and modularity.
- You can create custom decorators or use Python’s built-in decorators for specific use cases.