Brief Notes on Python Decorators

A Teaching Guide

Curated by ChatGPT

1. What is a Decorator?

- A decorator is a function that takes another function (or method) as input, adds extra functionality, and returns it.
- Used to wrap behavior around functions or classes without modifying them.
- Think of it as: Decorators = Wrappers around functions.

2. Why Use Decorators?

• Code Reusability – Apply the same functionality to multiple functions.

• Separation of Concerns – Keep core logic clean and handle extra behavior externally.

• Readability – Clear syntax using '@decorator' notation.

3. Syntax Example

```
def decorator(func):
    def wrapper(*args, **kwargs):
        # Pre-action
        result = func(*args, **kwargs)
        # Post-action
        return result
    return wrapper

@decorator
def my_function():
    print('Hello World')

my_function()
```

4. Types of Decorators

• Function Decorators – Used for standalone functions.

• Method Decorators – Used for class methods (e.g., @classmethod, @staticmethod).

• Class Decorators – Modify or enhance class behavior.

5. Built-in Decorators

• @staticmethod – Defines a static method (no self/cls).

• @classmethod – Defines a method bound to the class.

• @property – Turns a method into an attribute.

6. Real-World Examples

```
Example 1: Logging Decorator
def log(func):
    def wrapper(*args, **kwargs):
        print(f'Calling {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

@log
def greet(name):
    print(f'Hello {name}')

greet('Alice')

Example 2: Execution Time
import time
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f'{func.__name__} took {end-start:.4f}s')
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(2)

slow_function()
```

7. Advanced: Decorators with Arguments

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

@repeat(3)
def say_hi():
    print('Hi!')

say_hi()
```

8. Key Takeaways

• Decorators wrap functions with extra behavior.

• Useful for logging, timing, validation, or access control.

• Built-in decorators include @staticmethod, @classmethod, and @property.

• Parameterized decorators add flexibility (e.g., @repeat(3)).