Python Decorators - Complete Study Notes

What are Decorators?

A **decorator** is a design pattern in Python that allows you to modify or extend the functionality of functions or classes without permanently modifying their structure. Decorators wrap another function and can execute code before and after the wrapped function runs.

Key Points:

- Decorators are functions that take another function as an argument
- They return a modified version of the function
- Use `@decorator_name` syntax above the function definition
- Based on the concept of higher-order functions and closures

Basic Syntax

```
@decorator_function
def target_function():
    pass

# Equivalent to:
# target_function = decorator_function(target_function)
```

Simple Examples

1. Basic Decorator

```
def my_decorator(func):
    def wrapper():
        print("Before function execution")
        func()
        print("After function execution")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Before function execution
# Hello!
# After function execution
```

2. Decorator with Arguments

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
```

```
        print(f"Calling function: {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Function {func.__name__} finished")
        return result
    return wrapper

@my_decorator
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 3)
print(f"Result: {result}")
```

3. Timing Decorator

```
import time
from functools import wraps

def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(1)
    return "Done!"
```

Advanced Examples

1. Decorator with Parameters

```
def repeat(times):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
```

```
    print(f"Hello, {name}!")

greet("Alice")
# Output:
# Hello, Alice!
# Hello, Alice!
# Hello, Alice!
```

2. Class-based Decorator

```
class CountCalls:
    def __init__(self, func):
        self.func = func
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"Call {self.count} of {self.func.__name__}")
        return self.func(*args, **kwargs)

@CountCalls
def say_hello():
    print("Hello!")

say_hello()  # Call 1 of say_hello
say_hello()  # Call 2 of say_hello
```

3. Caching Decorator

```
from functools import wraps

def memoize(func):
    cache = {}

    @wraps(func)
    def wrapper(*args, **kwargs):
        key = str(args) + str(sorted(kwargs.items()))
        if key not in cache:
            cache[key] = func(*args, **kwargs)
            print(f"Computing {func.__name__}{args}")
        else:
            print(f"Using cached result for {func.__name__}{args}")
        return cache[key]
    return wrapper

@memoize
def fibonacci(n):
    if n < 2:
        return n
```

```
    return fibonacci(n-1) + fibonacci(n-2)
```

4. Authorization Decorator

```
def requires_auth(permission):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Simulate user check
            user_permissions = ['read', 'write']  # This would come from session/DB

            if permission not in user_permissions:
                raise PermissionError(f"Access denied. Required permission: {permissi

            return func(*args, **kwargs)
        return wrapper
    return decorator

@requires_auth('admin')
def delete_user(user_id):
    return f"User {user_id} deleted"
```

Built-in Decorators

1. @property

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def area(self):
        return 3.14159 * self._radius ** 2
```

2. @staticmethod and @classmethod

```
class MathUtils:
    class_variable = "I'm a class variable"
```

```
    @staticmethod
    def add(a, b):
        return a + b

    @classmethod
    def get_class_variable(cls):
        return cls.class_variable
```

Multiple Decorators

```
def bold(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return f"<b>{result}</b>"
    return wrapper

def italic(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return f"<i>{result}</i>"
    return wrapper

@bold
@italic
def say_hello(name):
    return f"Hello, {name}!"

print(say_hello("World"))  # <b><i>Hello, World!</i></b>
```

Important Concepts

1. functools.wraps - DETAILED EXPLANATION

The Problem: When you create a decorator, the wrapper function replaces the original function. This means all the metadata (name, docstring, annotations, etc.) of the original function gets lost.

Without @wraps (WRONG WAY):

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function")
        result = func(*args, **kwargs)
        print("After function")
        return result
    return wrapper

def calculate_area(radius):
    """Calculate area of a circle given radius."""
```

```
    return 3.14159 * radius * radius

print("BEFORE decoration:")
print(f"Name: {calculate_area.__name__}")
print(f"Doc: {calculate_area.__doc__}")
print(f"Module: {calculate_area.__module__}")

# Apply decorator
calculate_area = my_decorator(calculate_area)

print("\nAFTER decoration (WITHOUT @wraps):")
print(f"Name: {calculate_area.__name__}")      # 'wrapper' instead of 'calculate_are
print(f"Doc: {calculate_area.__doc__}")        # None instead of the docstring
print(f"Module: {calculate_area.__module__}")  # Wrong module info
```

Output:

```
BEFORE decoration:
Name: calculate_area
Doc: Calculate area of a circle given radius.
Module: __main__

AFTER decoration (WITHOUT @wraps):
Name: wrapper
Doc: None
Module: __main__
```

With @wraps (CORRECT WAY):

```
from functools import wraps

def my_decorator(func):
    @wraps(func)  # This line preserves original function metadata
    def wrapper(*args, **kwargs):
        print("Before function")
        result = func(*args, **kwargs)
        print("After function")
        return result
    return wrapper

@my_decorator
def calculate_area(radius):
    """Calculate area of a circle given radius."""
    return 3.14159 * radius * radius

print("AFTER decoration (WITH @wraps):")
print(f"Name: {calculate_area.__name__}")      # 'calculate_area' - PRESERVED!
print(f"Doc: {calculate_area.__doc__}")        # Original docstring - PRESERVED!
print(f"Module: {calculate_area.__module__}")  # Correct module - PRESERVED!
```

Output:

```
AFTER decoration (WITH @wraps):
Name: calculate_area
Doc: Calculate area of a circle given radius.
Module: __main__
```

What Metadata Gets Preserved? `@wraps(func)` copies these attributes from the original function to the wrapper:

```python
from functools import wraps

def show_metadata(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

@show_metadata
def example_function(x: int, y: str = "default") -> str:
    """
    This is an example function with type hints and default values.

    Args:
        x: An integer parameter
        y: A string parameter with default value

    Returns:
        A formatted string
    """
    return f"x={x}, y={y}"

# All these are preserved:
print(f"__name__: {example_function.__name__}")           # example_function
print(f"__doc__: {example_function.__doc__}")             # The full docstring
print(f"__module__: {example_function.__module__}")       # Module name
print(f"__qualname__: {example_function.__qualname__}")   # Qualified name
print(f"__annotations__: {example_function.__annotations__}") # Type hints
print(f"__dict__: {example_function.__dict__}")           # Function attributes
```

Why This Matters - Real-World Problems: 1. Documentation Tools Break:

```python
# Without @wraps
def bad_decorator(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

@bad_decorator
```

```python
def important_function():
    """This function does something important."""
    pass

help(important_function)  # Shows help for 'wrapper', not 'important_function'
# Documentation tools like Sphinx will be confused
```

2. Debugging Becomes Harder:

```python
import traceback

def problematic_decorator(func):
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception:
            traceback.print_exc()  # Will show 'wrapper' in traceback, not original
            raise
    return wrapper
```

3. Function Introspection Fails:

```python
# Code that relies on function names
function_registry = {}

def register_function(func):
    function_registry[func.__name__] = func
    return func

def bad_decorator(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

@register_function
@bad_decorator  # Without @wraps
def my_function():
    pass

print(function_registry)  # {'wrapper': <function>} instead of {'my_function': <func
```

4. Testing and Mocking Issues:

```python
import unittest.mock

def cache_decorator(func):
    cache = {}
    def wrapper(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cache:
            cache[key] = func(*args, **kwargs)
```

```
        return cache[key]
    return wrapper

@cache_decorator  # Without @wraps
def get_user_data(user_id):
    # Expensive API call
    return f"Data for user {user_id}"

# This won't work as expected in tests:
with unittest.mock.patch('__main__.get_user_data') as mock:
    # Mock might not work because the function name is now 'wrapper'
    pass
```

Complete Example Showing the Difference:

```
from functools import wraps

# BAD: Without @wraps
def bad_timing_decorator(func):
    import time
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Function took {end - start:.4f} seconds")
        return result
    return wrapper

# GOOD: With @wraps
def good_timing_decorator(func):
    import time
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")  # Can use original
        return result
    return wrapper

@bad_timing_decorator
def slow_function_bad():
    """A slow function for testing."""
    import time
    time.sleep(0.1)

@good_timing_decorator
def slow_function_good():
```

```
    """A slow function for testing."""
    import time
    time.sleep(0.1)

print("=== BAD DECORATOR ===")
print(f"Name: {slow_function_bad.__name__}")    # wrapper
print(f"Doc: {slow_function_bad.__doc__}")      # None
slow_function_bad()

print("\n=== GOOD DECORATOR ===")
print(f"Name: {slow_function_good.__name__}")   # slow_function_good
print(f"Doc: {slow_function_good.__doc__}")     # A slow function for testing.
slow_function_good()
```

Key Takeaway: Always use `@wraps(func)` in your decorators to maintain the original function's identity and make your code more maintainable, debuggable, and compatible with other tools.

2. Decorator Factory Pattern

```
def decorator_factory(arg1, arg2):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Use arg1, arg2 here
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

Tricky Interview Questions

Question 1: What's wrong with this decorator?

```
def broken_decorator(func):
    def wrapper():
        print("Before")
        func()
        print("After")
    return wrapper

@broken_decorator
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")  # This will cause an error!
```

Answer: The wrapper function doesn't accept arguments, but the decorated function `greet` expects a `name` parameter. Fix: Use `*args, **kwargs`.

Question 2: Explain the output

```
def decorator_factory(msg):
    def decorator(func):
        def wrapper(*args, **kwargs):
            print(msg)
            return func(*args, **kwargs)
        return wrapper
    return decorator

@decorator_factory("Hello")
@decorator_factory("World")
def test():
    print("Test function")

test()
```

Answer: Output will be:

```
World
Hello
Test function
```

Decorators are applied bottom-up (inside-out).

Question 3: What happens here?

```
def my_decorator(func):
    print(f"Decorating {func.__name__}")
    def wrapper():
        print("Wrapper executed")
        return func()
    return wrapper

print("Before decoration")

@my_decorator
def test_func():
    print("Original function")

print("After decoration")
test_func()
```

Answer: Output:

```
Before decoration
Decorating test_func
After decoration
Wrapper executed
Original function
```

The decorator function runs at **import/definition time**, not when the function is called.

Question 4: Implement a retry decorator

```
def retry(max_attempts=3, delay=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts - 1:
                        raise e
                    print(f"Attempt {attempt + 1} failed: {e}")
                    time.sleep(delay)
        return wrapper
    return decorator

@retry(max_attempts=3, delay=0.5)
def unreliable_function():
    import random
    if random.random() < 0.7:
        raise Exception("Random failure!")
    return "Success!"
```

Question 5: Create a decorator that caches results but expires after a certain time

```
import time
from functools import wraps

def timed_cache(expiry_seconds=60):
    def decorator(func):
        cache = {}

        @wraps(func)
        def wrapper(*args, **kwargs):
            key = str(args) + str(sorted(kwargs.items()))
            current_time = time.time()

            if key in cache:
                result, timestamp = cache[key]
                if current_time - timestamp < expiry_seconds:
                    print("Cache hit")
                    return result
                else:
                    print("Cache expired")
                    del cache[key]

            print("Computing new result")
```

```
            result = func(*args, **kwargs)
            cache[key] = (result, current_time)
            return result

        return wrapper
    return decorator
```

Question 6: What's the difference between these two?

```
# Version 1
@my_decorator
def func1():
    pass

# Version 2
def func2():
    pass
func2 = my_decorator(func2)
```

Answer: They are functionally identical. The `@` syntax is just syntactic sugar for the second form.

Question 7: How to access the original function from a decorated function?

```
def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)

    wrapper.__wrapped__ = func  # Store reference to original
    return wrapper

@my_decorator
def original_func():
    return "Original"

# Access original function
original_func.__wrapped__()  # Calls original function directly
```

Common Pitfalls

1. **Not using `*args, **kwargs`** - Decorator won't work with functions that have parameters
2. **Not using `@wraps(func)`** - Loses original function metadata
3. **Modifying mutable default arguments** in decorator factories
4. **Not handling return values** properly
5. **Confusing decorator execution time** (definition vs call time)

Best Practices

1. Always use `@functools.wraps(func)`
2. Use `*args, **kwargs` for maximum flexibility
3. Keep decorators simple and focused on single responsibility
4. Document decorator behavior clearly
5. Consider using classes for stateful decorators
6. Test decorators thoroughly with different function signatures