decorators

Write a decorator that counts how many times a function is called."

```python
def call_counter(func):
    count = 0   # counter stored in closure

    def wrapper(*args, **kwargs):
        nonlocal count  # allows modifying the outer variable
        count += 1
        print(f"{func.__name__} has been called {count} times")
        return func(*args, **kwargs)

    return wrapper


# Example usage:
@call_counter
def greet(name):
    print(f"Hello, {name}!")


greet("Alice")
greet("Bob")
greet("Charlie")
```

Output:

```
greet has been called 1 times
Hello, Alice!
greet has been called 2 times
Hello, Bob!
greet has been called 3 times
Hello, Charlie!
```

Python Decorators

Introduction to Decorators

- **What are Decorators?**

    - Decorators are a design pattern in Python that allows you to add new functionality to an existing object (like a function, method, or class) without modifying its structure.
    - They are essentially functions that wrap other functions, methods, or classes.
    - Decorators use the `@` symbol followed by the decorator name, placed above the definition of the function or class being decorated.
    - Key concept: Functions in Python are first-class objects—they can be passed as arguments, returned from other functions, and assigned to variables.
- **Why Use Decorators?**
    - Promote code reuse and modularity.
    - Separate concerns (e.g., core logic vs. logging, timing, or authentication).
    - Make code cleaner and more readable by avoiding repetitive boilerplate.
- **How Decorators Work Under the Hood**
    - A decorator is a function that takes another function as an argument, adds some functionality, and returns a new function (the wrapper).
    - When you apply `@decorator` to a function `func`, it's equivalent to `func = decorator(func)`.
    - The wrapper function typically calls the original function and adds behavior before/after it.
- **Syntax Basics**
    - Define a decorator function that accepts a callable (e.g., function) and returns a wrapper.
    - Use `*args` and `**kwargs` in the wrapper to handle any arguments passed to the original function.
    - Preserve metadata (name, docstring) using `functools.wraps` to avoid issues like incorrect function names in debugging.

Key Concepts for Interview Preparation

- **Interview Tip:** Be ready to explain decorators as "syntactic sugar" for function wrapping. Know that they execute at definition time (when the module is loaded), not at runtime.
- **Common Pitfalls:**
    - Forgetting to return the wrapper function.
    - Not handling arguments properly (use `*args, **kwargs`).
    - Losing original function metadata (fix with `@wraps`).
- **Advanced Notes:**
    - Decorators can be stacked (multiple `@` lines; applied from bottom to top).
    - Can decorate classes or methods.
    - Can take arguments themselves (requires a decorator factory—a function returning the actual decorator).

Example 1: Simple Logging Decorator

- **Scenario:** Add logging to track when a function is called and its arguments.
- **Use Case:** Debugging, auditing function calls in production code without altering the function itself (e.g., in web apps to log API requests).

```
import functools
```

```
def log_decorator(func):
    @functools.wraps(func)  # Preserves original function's name and docstring
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args: {args} and kwargs: {kwargs}")
        result = func(*args, **kwargs)  # Call the original function
        print(f"{func.__name__} returned: {result}")
        return result
    return wrapper

@log_decorator  # Applying the decorator
def add_numbers(a, b):
    """Adds two numbers."""
    return a + b

# Usage
result = add_numbers(3, 5)
# Output:
# Calling add_numbers with args: (3, 5) and kwargs: {}
# add_numbers returned: 8
```

- **Explanation Step-by-Step:**
  1. Define `log_decorator` which takes `func`.
  2. Inside, define `wrapper` that logs before/after calling `func`.
  3. Use `@functools.wraps` to keep `add_numbers`'s identity.
  4. Apply `@log_decorator` to `add_numbers`.
  5. When called, it logs entry/exit without changing the function's core logic.

Example 2: Timing Decorator

- **Scenario:** Measure execution time of a function.
- **Use Case:** Performance optimization, profiling slow functions in data processing or machine learning pipelines.

```
import time
import functools

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

@timer_decorator
```

```
def slow_function(n):
    """Simulates a slow computation."""
    time.sleep(n)  # Sleep for n seconds
    return f"Slept for {n} seconds"

# Usage
result = slow_function(2)
# Output example:
# slow_function took 2.0010 seconds
```

- **Explanation Step-by-Step:**
  1. `timer_decorator` wraps the function.
  2. `wrapper` times the execution using `time.time()`.
  3. Apply to `slow_function` to add timing without modifying it.
  4. Useful for benchmarking; can be removed easily if not needed.

Use Case Scenarios

- **Logging and Debugging:** Track function calls in large applications (e.g., Flask/Django for request logging).
- **Authentication/Authorization:** Check user permissions before executing a function (e.g., `@login_required` in web frameworks).
- **Caching/Memoization:** Store results of expensive functions to avoid recomputation (e.g., `@lru_cache` from `functools`).
- **Rate Limiting:** Limit how often a function can be called (e.g., API endpoints).
- **Error Handling:** Wrap functions to catch and handle exceptions uniformly.
- **Performance Monitoring:** Time functions or count calls in production.
- **Interview Tip:** Mention real-world examples like Flask's `@app.route` or Django's `@csrf_exempt` to show practical knowledge.

Quick Review for Interviews

- **Define:** "A decorator is a function that modifies another function's behavior by wrapping it."
- **Syntax:** `@decorator_name` above the function.
- **Implementation Steps:** Decorator func → wrapper func → call original → return wrapper.
- **Pros:** Reusability, clean code.
- **Cons:** Can make debugging harder if overused; adds indirection.
- **Practice Question:** "Write a decorator that counts how many times a function is called." (Hint: Use a counter in the decorator.)

```
def call_counter(func):
    count = 0   # counter stored in closure

    def wrapper(*args, **kwargs):
        nonlocal count  # allows modifying the outer variable
        count += 1
        print(f"{func.__name__} has been called {count} times")
```

```
        return func(*args, **kwargs)

    return wrapper


# Example usage:
@call_counter
def greet(name):
    print(f"Hello, {name}!")


greet("Alice")
greet("Bob")
greet("Charlie")
```

Output:

```
greet has been called 1 times
Hello, Alice!
greet has been called 2 times
Hello, Bob!
greet has been called 3 times
Hello, Charlie!
```