4. Apply Python Decorators for Reusable Function Enhancements

Why: Decorators allow you to add functionality (e.g., logging, timing) to functions without modifying their core logic, promoting DRY (Don't Repeat Yourself) principles. This is vital for clean, modular code in large projects, enhancing debugging and performance monitoring.
How to Implement: Define a decorator function that takes a function and returns a wrapped version. Use `@decorator_name` syntax above target functions. Ideal for APIs, where you might add authentication or rate-limiting universally.

```
python
# Without decorator
def add(a, b):
    return a + b

# Hack: Timing decorator
import time
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Execution time: {end - start} seconds")
        return result
    return wrapper

@timer
def add(a, b):
    time.sleep(1)  # Simulate delay
    return a + b

print(add(5, 3))  # Outputs result and time
```

Analysis: Decorators separate concerns, making code testable and extensible. In profiling, they reveal bottlenecks without invasive changes; for instance, adding logging across 100 functions becomes trivial, saving hours in maintenance.