Wali Muhammad
@full-stackengineer

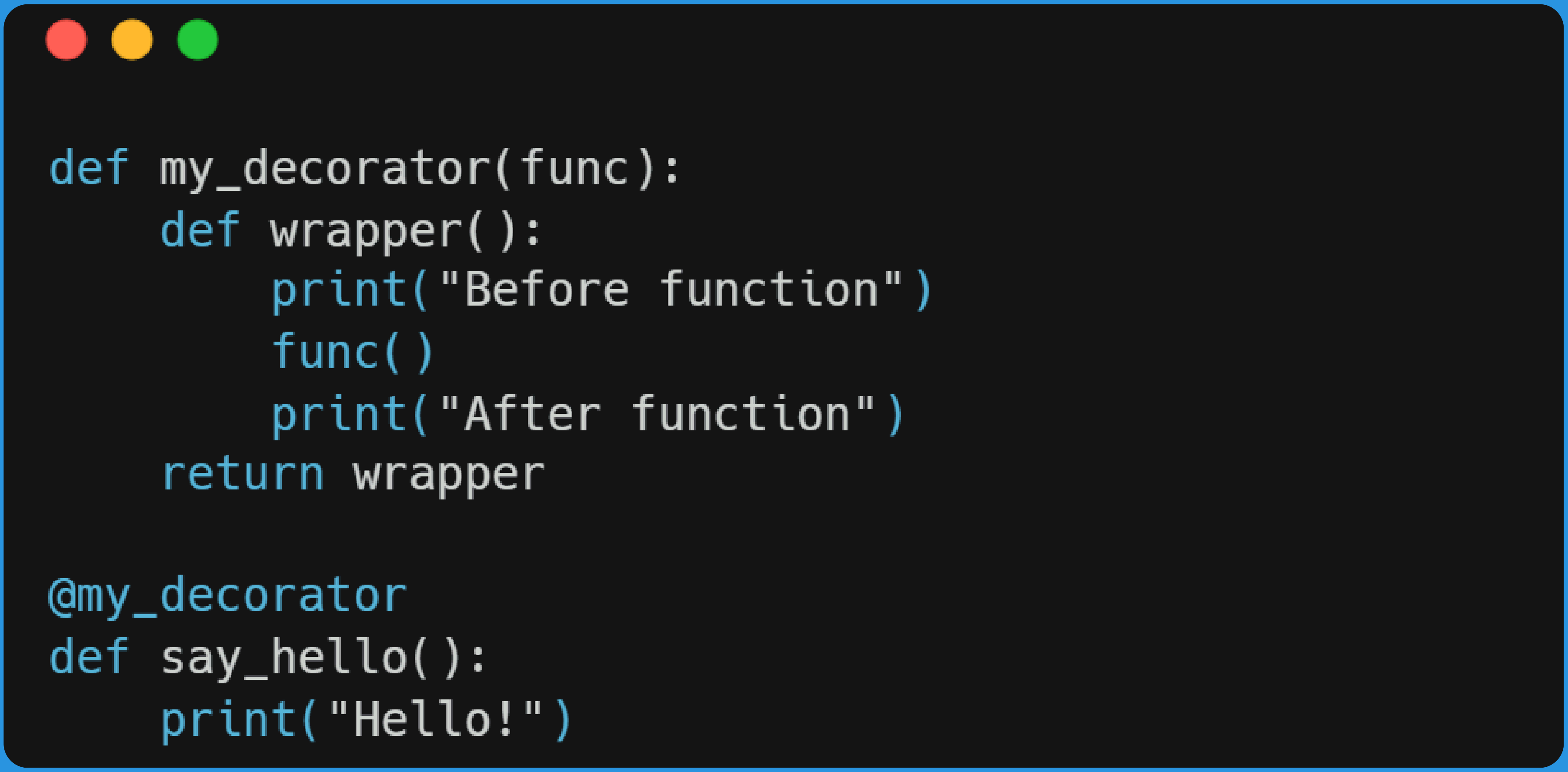
Unlock Python’s Hidden Power
Decorators & Generators

Wali Muhammad
@full-stackengineer

What Are **Decorators?**

A decorator is a design pattern that modifies a function or class without altering its code. Think of it as wrapping a function with additional functionality.

Basic Decorator **Example**: Enhancing a Function



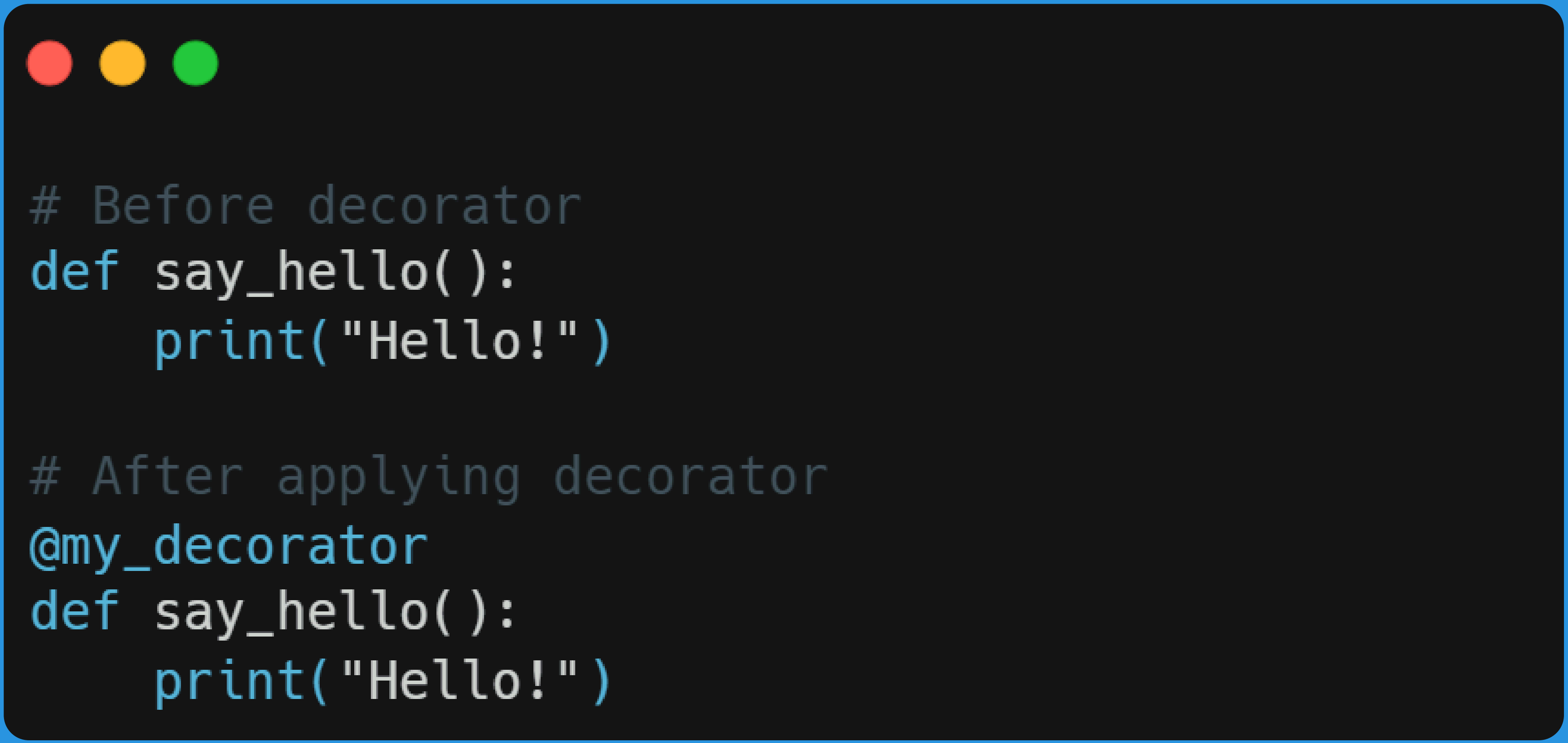
```python
def my_decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper

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

Wali Muhammad
@full-stackengineer

Step-by-Step: How Decorators Wrap Functions

- Define a decorator that accepts a function.
- Wrap it in another function to add behavior.
- Apply it using **‘@decorator_name’**.

Before and After: Applying a Decorator to a Function



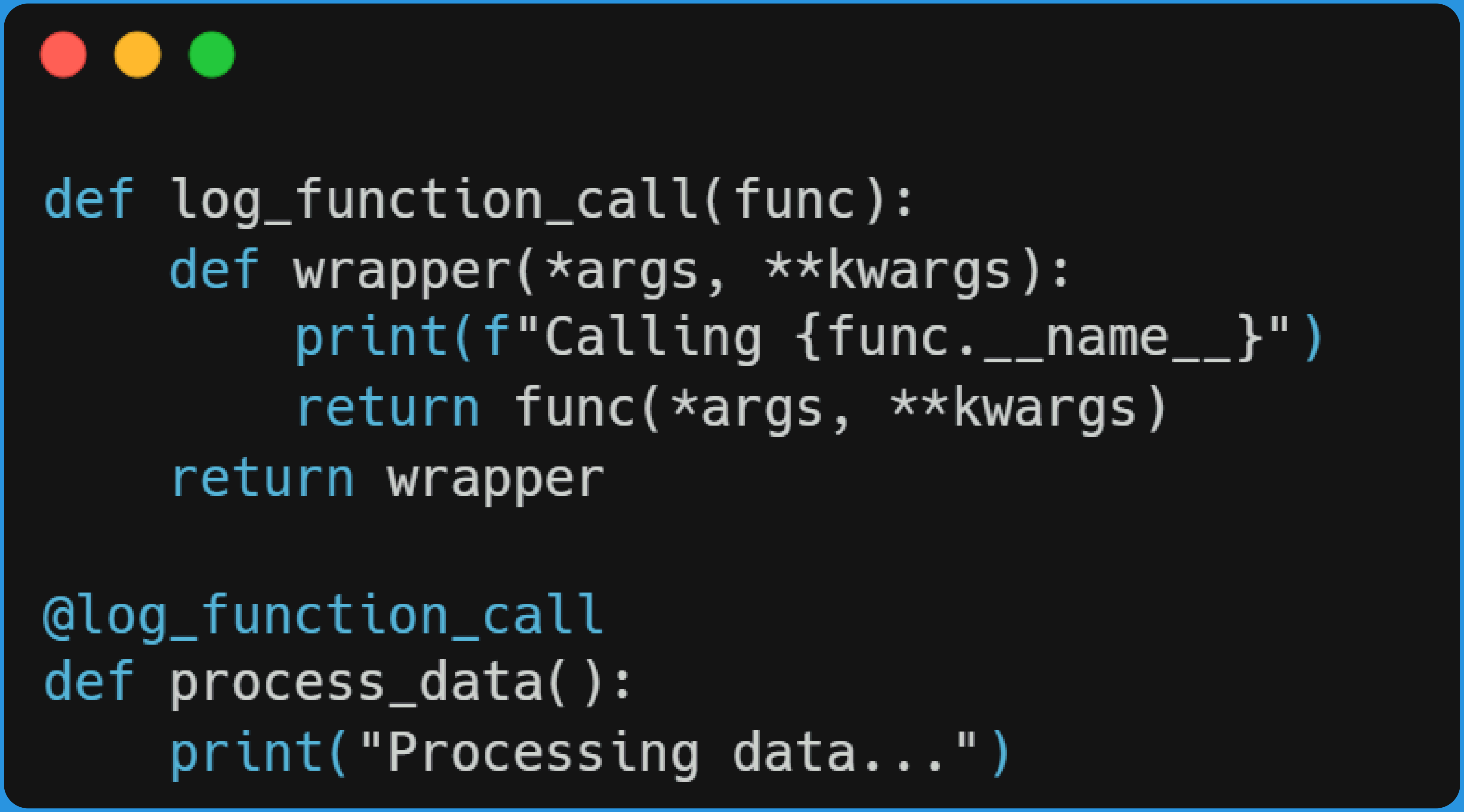
```
# Before decorator
def say_hello():
    print("Hello!")

# After applying decorator
@my_decorator
def say_hello():
    print("Hello!")
```

Wali Muhammad
@full-stackengineer

Real-World Applications of Decorators

- **Logging:** Track function execution.
- **Authentication:** Verify permissions.
- **Caching:** Optimize repeated operations.

Logging Decorator: Tracking Function Execution



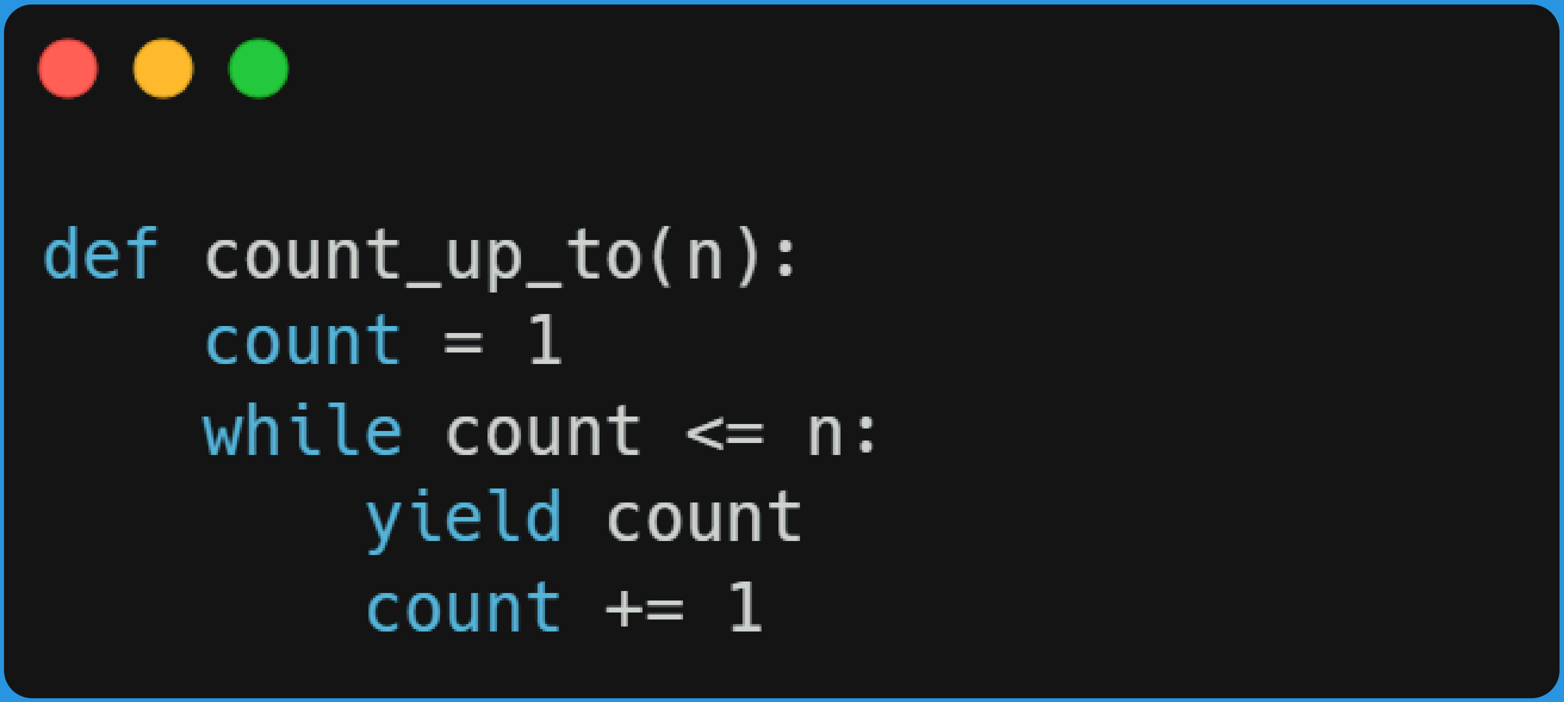
```
def log_function_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log_function_call
def process_data():
    print("Processing data...")
```

Wali Muhammad
@full-stackengineer

Generators: Efficient Iteration Made Simple

A generator produces values one at a time, simplifying iteration and saving memory—perfect for handling large datasets or infinite sequences.

Simple Generator Example: Counting Up to a Number



```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1
```

Wali Muhammad
@full-stackengineer

Generators: Efficient Iteration Made Simple

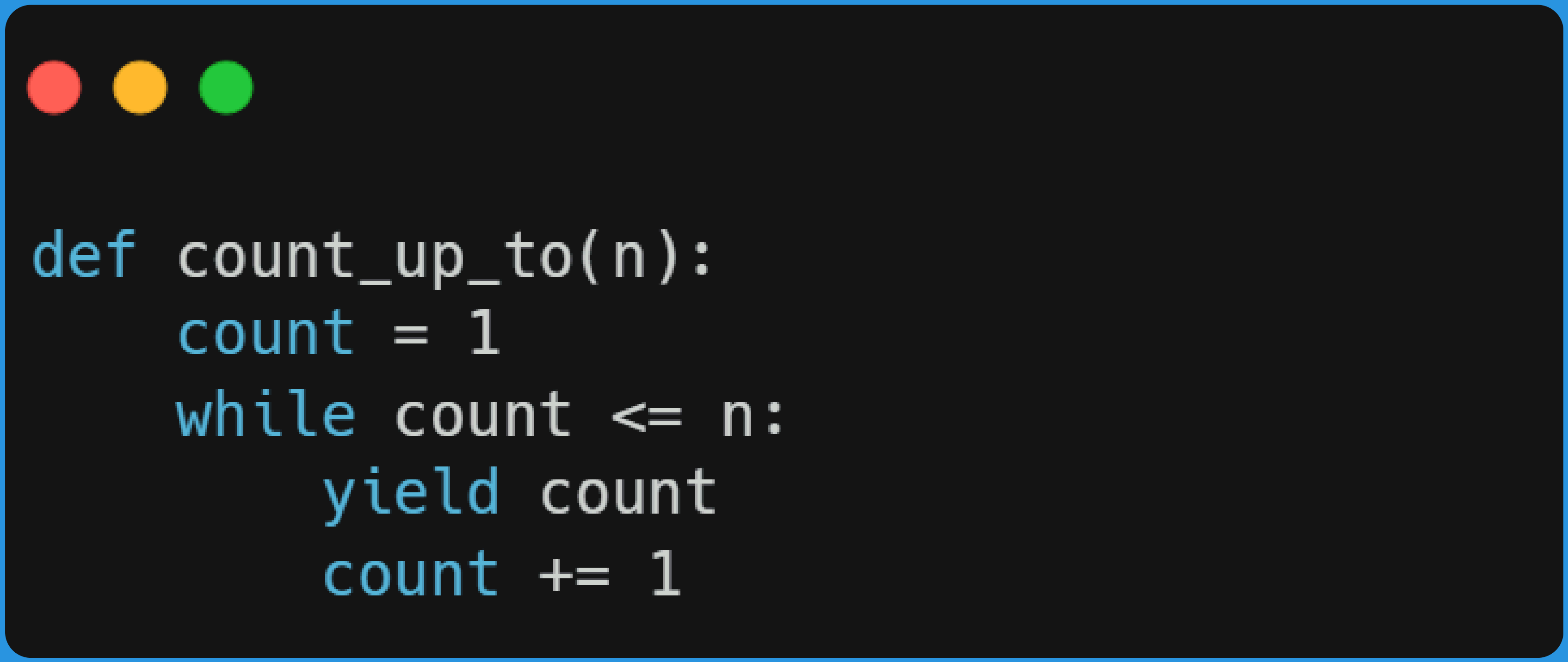
- Define with **yield** instead of **return**.
- Generates values **lazily** as needed.
- Ideal for memory efficiency.

Wali Muhammad
@full-stackengineer

Practical Generator Use Cases

- **Large datasets:** Process one item at a time.
- **Infinite sequences:** Count without limits.
- **Lazy evaluation:** Compute only when necessary.

Generator for Large Files: Processing One Line at a Time



```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1
```

Wali Muhammad

@full-s tackengineer

Decorators and generators streamline your code and boost efficiency.

Start using them to take your Python skills to the next level!