Python

Python Decorators

Enhancing Functions with Elegant Metaprogramming

March 17, 2025

Source Code

1. Introduction to Python Decorators

Python decorators are a powerful feature that allow developers to modify or enhance functions and classes without changing their core implementation. In essence, decorators are a design pattern that lets you "wrap" one function with another function to extend its behavior.

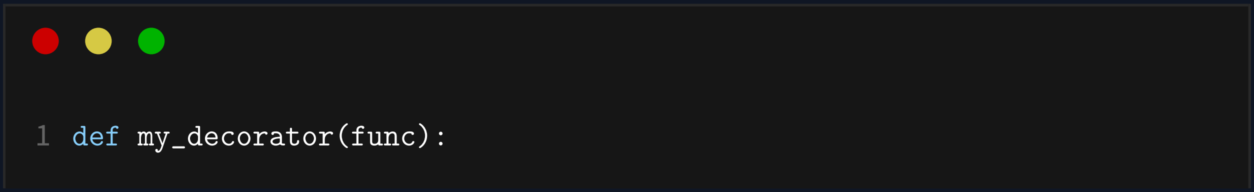
1.1. What Are Decorators?

At their core, decorators are a form of metaprogramming – code that manipulates other code. They provide a clean syntax to modify the behavior of functions or classes using the **@** symbol.

- **Higher-Order Functions:** Functions that take another function as an argument
- **Syntactic Sugar:** The **@decorator** syntax is equivalent to **function = decorator(function)**
- **Non-Invasive:** Add functionality without modifying the original code
- **Reusability:** Apply the same behavior across multiple functions

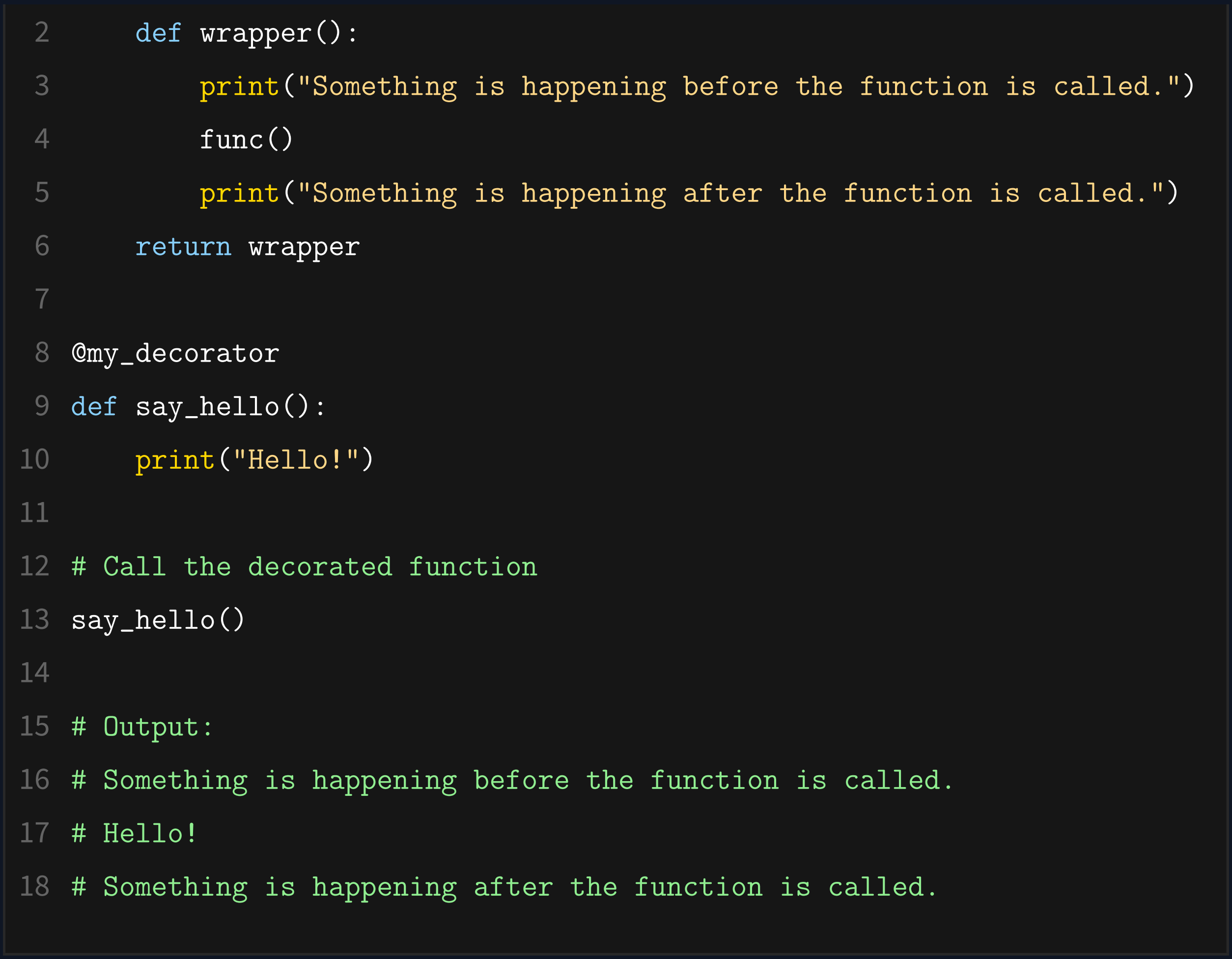
2. Basic Decorator Pattern

The fundamental decorator pattern consists of a function that takes another function as input and returns a new function with enhanced behavior:



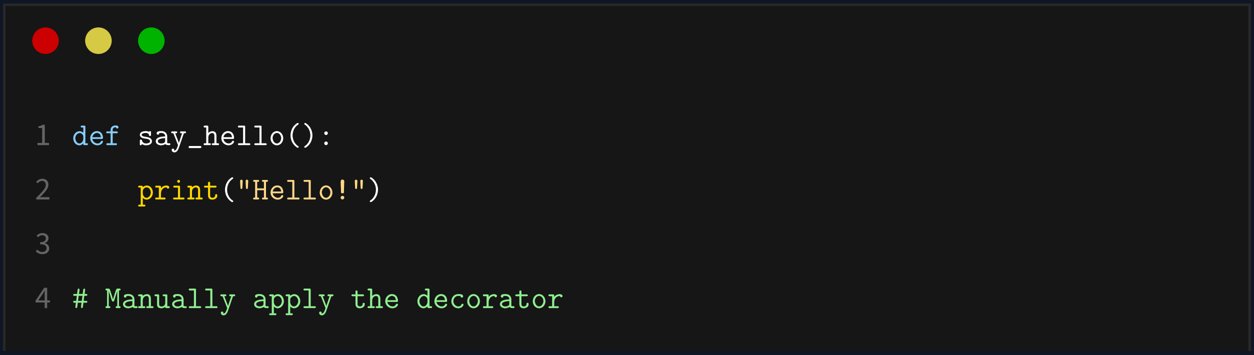
```python
def my_decorator(func):
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com



```python
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

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

# Call the decorated function
say_hello()

# Output:
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

The **@my_decorator** syntax is equivalent to:



```python
def say_hello():
    print("Hello!")

# Manually apply the decorator
```

```python
say_hello = my_decorator(say_hello)
```

3. Decorating Functions with Arguments

Real-world functions often have arguments. Decorators need to handle these arguments correctly:

```python
import functools

def decorator_with_args(func):
    @functools.wraps(func)  # Preserves the original function’s metadata
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with arguments: {args}, {kwargs}")
        result = func(*args, **kwargs)
        print(f"Function {func.__name__} returned: {result}")
        return result
    return wrapper

@decorator_with_args
def add(a, b):
    """Add two numbers."""
```

```python
    return a + b

# Call the decorated function
result = add(3, 5)
print(f"Result: {result}")

# Output:
# Calling add with arguments: (3, 5), {}
# Function add returned: 8
# Result: 8

# Check that metadata is preserved
print(add.__name__)  # ’add’ (not ’wrapper’)
print(add.__doc__)   # ’Add two numbers.’
```

4. Decorators with Parameters

Sometimes we need to create decorators that accept their own parameters:

```python
def repeat(times=2):
    """A decorator that runs a function multiple times"""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            result = None
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(times=3)
def greet(name):
    print(f"Hello, {name}!")
    return name

# Call the decorated function
greet("World")

# Output:
# Hello, World!
# Hello, World!
# Hello, World!
```

Note the triple-level nesting required for parameterized decorators:

- **Level 1: repeat()** - handles decorator parameters
- **Level 2: decorator()** - accepts the function being decorated
- **Level 3: wrapper()** - handles the function’s arguments

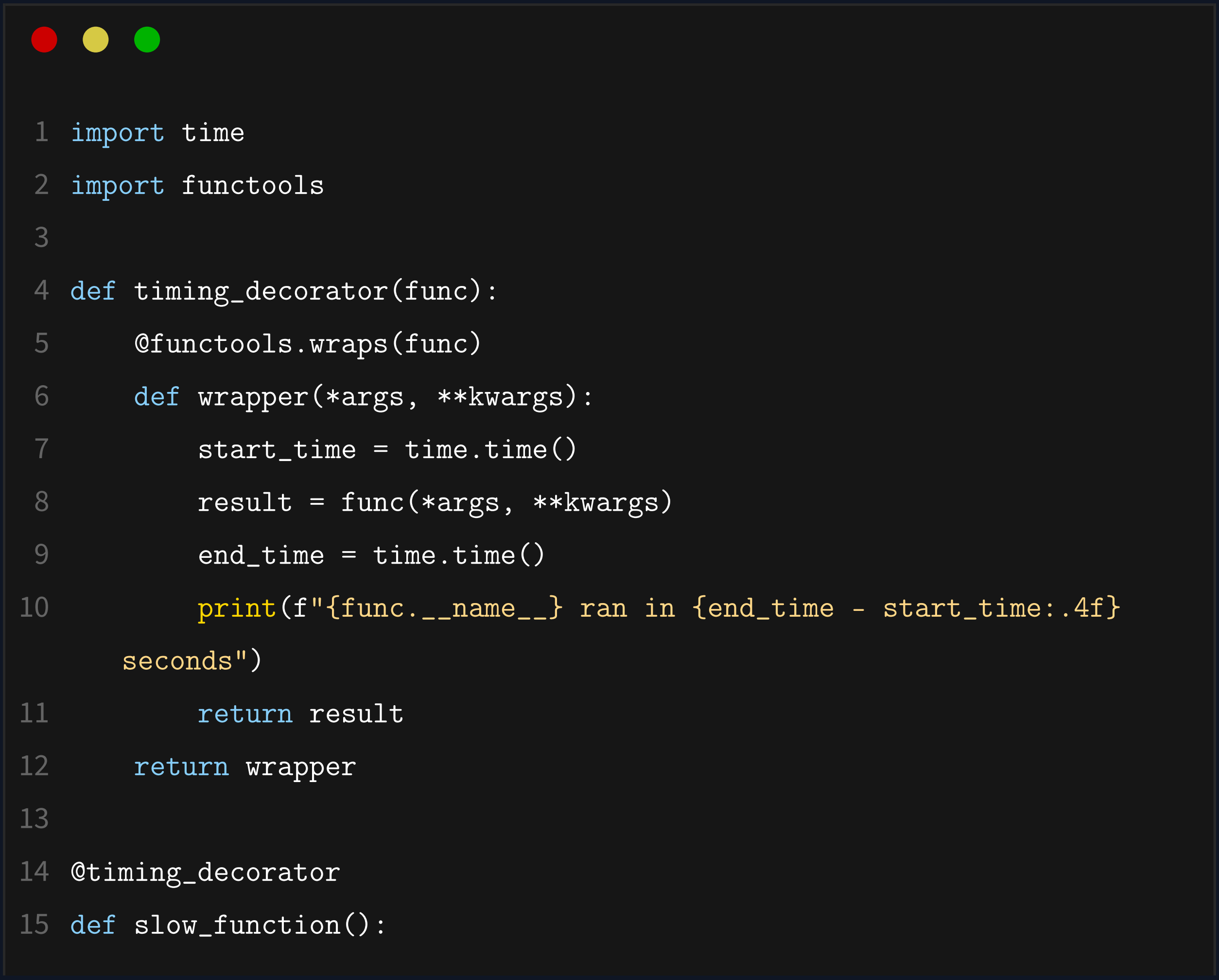
Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

5. Practical Applications

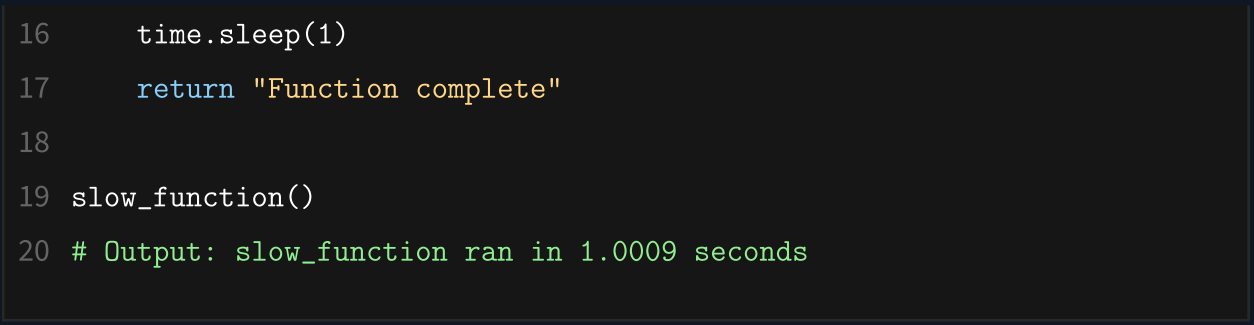
Decorators shine in many real-world scenarios where they help separate cross-cutting concerns from business logic.

5.1. Timing Functions

Measuring execution time without cluttering your functions:



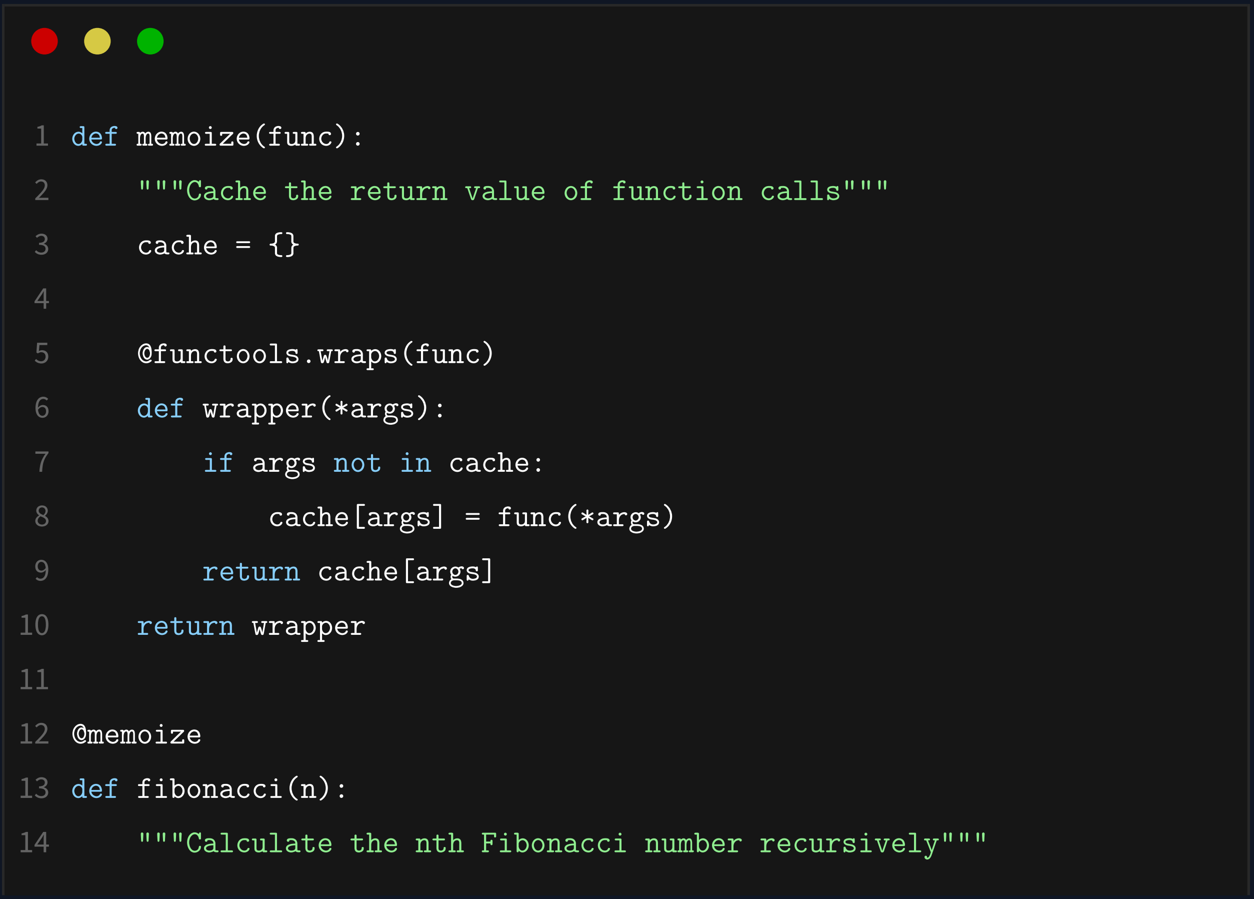
```python
import time
import functools

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

@timing_decorator
def slow_function():
```



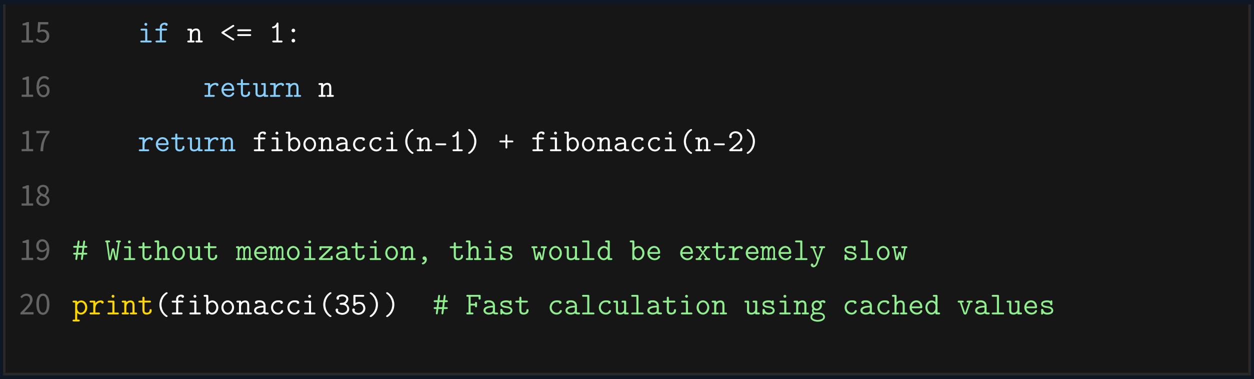
```python
    time.sleep(1)
    return "Function complete"

slow_function()
# Output: slow_function ran in 1.0009 seconds
```

5.2. Caching Results

Improve performance by storing previously calculated results:



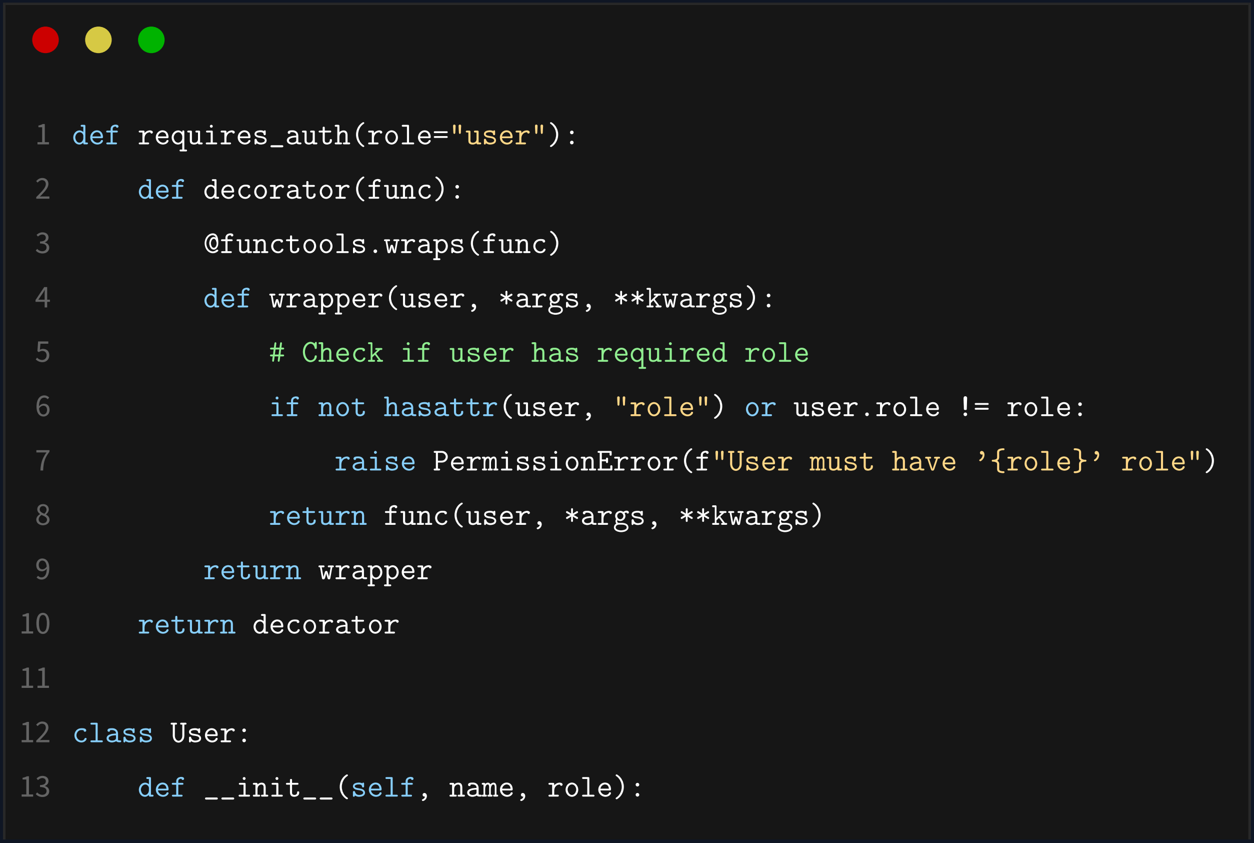
```python
def memoize(func):
    """Cache the return value of function calls"""
    cache = {}

    @functools.wraps(func)
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def fibonacci(n):
    """Calculate the nth Fibonacci number recursively"""
```



```python
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Without memoization, this would be extremely slow
print(fibonacci(35))  # Fast calculation using cached values
```

5.3. Authentication and Authorization

Control access to functions based on user roles:



```python
def requires_auth(role="user"):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(user, *args, **kwargs):
            # Check if user has required role
            if not hasattr(user, "role") or user.role != role:
                raise PermissionError(f"User must have '{role}' role")
            return func(user, *args, **kwargs)
        return wrapper
    return decorator

class User:
    def __init__(self, name, role):
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
🌐 www.asanchezyali.com

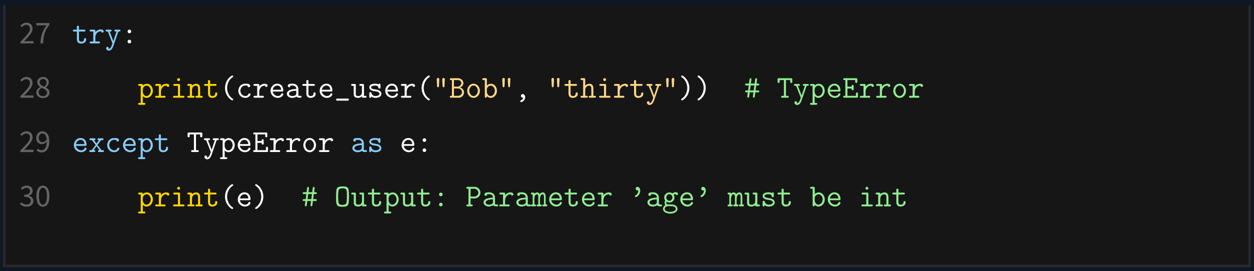
```python
        self.name = name
        self.role = role

@requires_auth(role="admin")
def delete_item(user, item_id):
    print(f"User {user.name} deleted item {item_id}")

# Admin user can delete items
admin = User("Alice", "admin")
delete_item(admin, 42)

# Regular user will get an error
regular_user = User("Bob", "user")
try:
    delete_item(regular_user, 42)
except PermissionError as e:
    print(e)  # Output: User must have ’admin’ role
```

5.4. Validation and Type Checking

Ensure function inputs meet requirements:

```python
def validate_types(**param_types):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # Get function parameter names
            import inspect
            sig = inspect.signature(func)
            bound_args = sig.bind(*args, **kwargs)

            # Check each parameter type
            for param_name, param_type in param_types.items():
                if param_name in bound_args.arguments:
                    value = bound_args.arguments[param_name]
                    if not isinstance(value, param_type):
                        raise TypeError(
                            f"Parameter ’{param_name}’ must be {param_type.__name__}"
                        )
            return func(*args, **kwargs)
        return wrapper
    return decorator

@validate_types(name=str, age=int)
def create_user(name, age):
    return f"User {name}, age {age} created"

print(create_user("Alice", 30))  # Works
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
🌐 www.asanchezyali.com



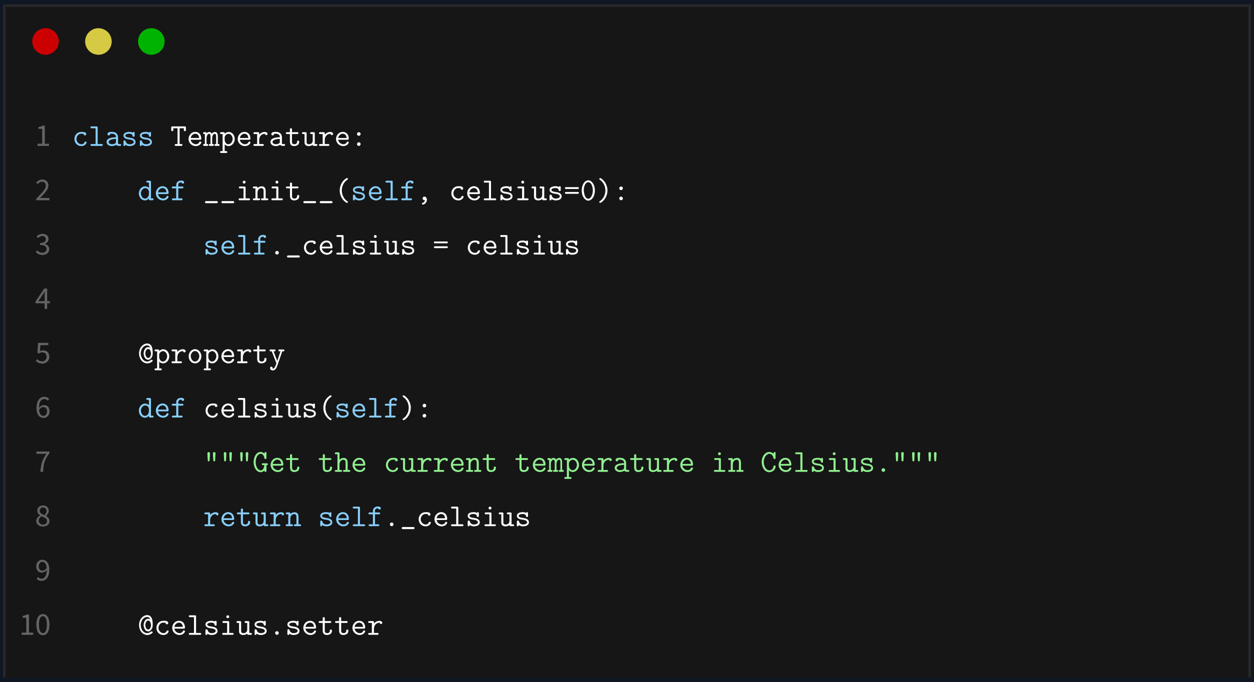
```python
try:
    print(create_user("Bob", "thirty"))  # TypeError
except TypeError as e:
    print(e)  # Output: Parameter ’age’ must be int
```

6. Built-in Decorators

Python includes several built-in decorators that demonstrate the power of this pattern.

6.1. Property Decorator

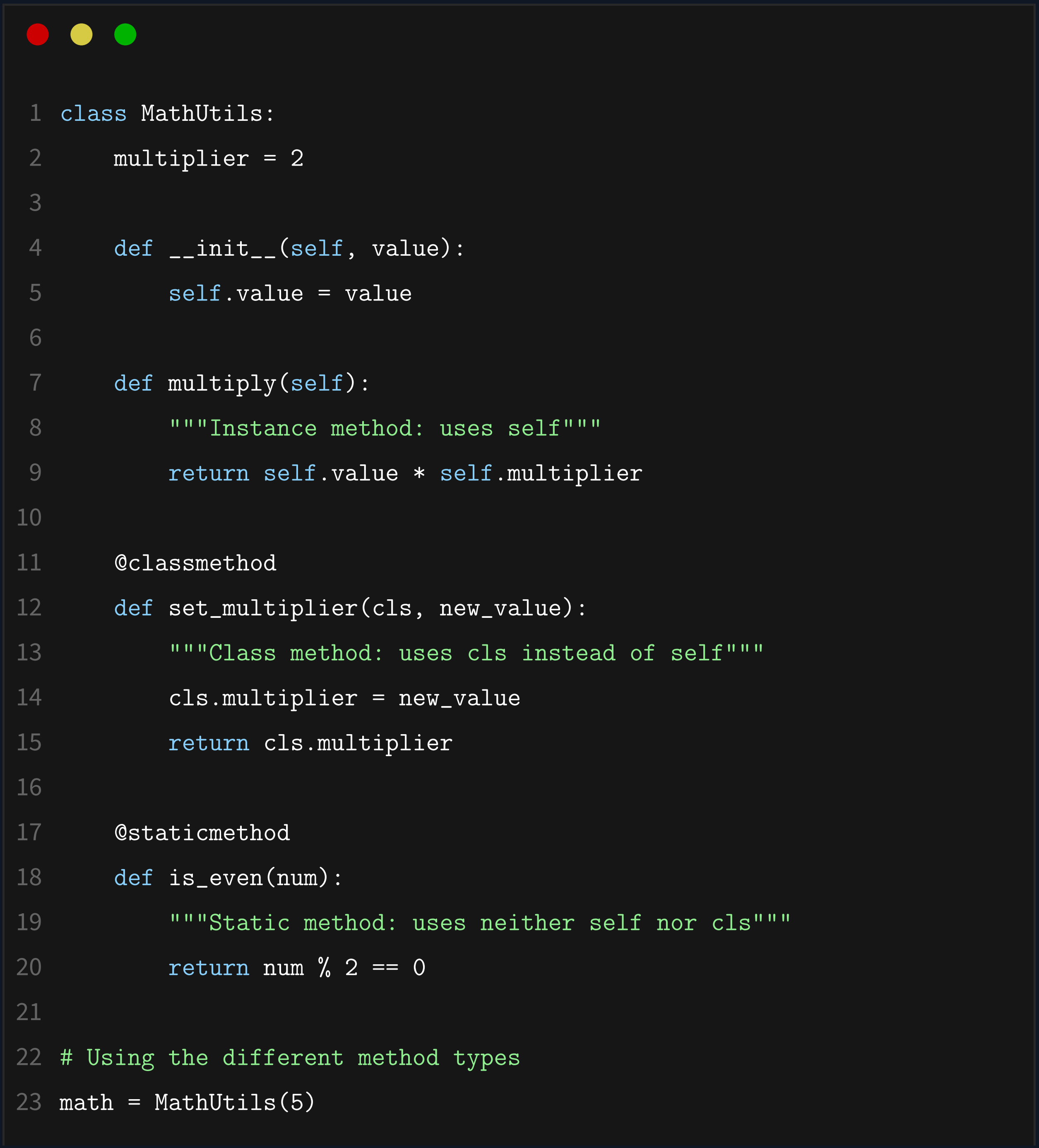
The **@property** decorator transforms methods into attribute-like accessors:



```python
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        """Get the current temperature in Celsius."""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature below absolute zero!")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Get the current temperature in Fahrenheit."""
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5/9

# Using the properties
temp = Temperature()
temp.celsius = 25
print(f"{temp.celsius} C is {temp.fahrenheit} F")

# Setting in Fahrenheit automatically updates Celsius
temp.fahrenheit = 68
print(f"{temp.fahrenheit} F is {temp.celsius} C")
```

6.2. Class and Static Method Decorators

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com



```python
class MathUtils:
    multiplier = 2

    def __init__(self, value):
        self.value = value

    def multiply(self):
        """Instance method: uses self"""
        return self.value * self.multiplier

    @classmethod
    def set_multiplier(cls, new_value):
        """Class method: uses cls instead of self"""
        cls.multiplier = new_value
        return cls.multiplier

    @staticmethod
    def is_even(num):
        """Static method: uses neither self nor cls"""
        return num % 2 == 0

# Using the different method types
math = MathUtils(5)
```

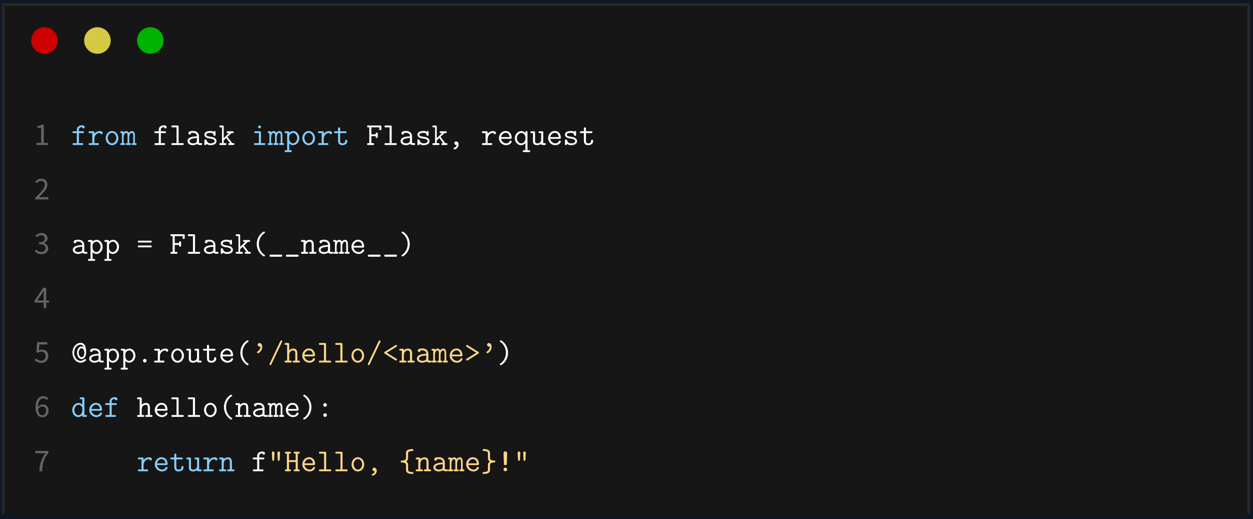
```python
print(math.multiply())  # 10 (5 * 2)

# Class method affects all instances
MathUtils.set_multiplier(3)
print(math.multiply())  # 15 (5 * 3)

# Static method is independent
print(MathUtils.is_even(4))  # True
```

7. Decorators in the Wild

Decorators are widely used in popular Python frameworks and libraries.

7.1. Flask Web Framework

Flask uses decorators for route definitions:



```python
from flask import Flask, request

app = Flask(__name__)

@app.route(’/hello/<name>’)
def hello(name):
    return f"Hello, {name}!"
```

```python

@app.route('/login', methods=['POST'])
def login():
    username = request.form['username']
    password = request.form['password']
    # Authentication logic here
    return f"Welcome back, {username}!"
```

7.2. Django Framework

Django uses decorators for views and authentication:

```python
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from django.views.decorators.http import require_POST

@login_required
def profile(request):
    # Only accessible to logged-in users
    return render(request, 'profile.html')

@require_POST
def update_profile(request):
    # Only accepts POST requests
```

```
    # Update profile logic
    return render(request, ’profile_updated.html’)
```

8. Best Practices

Follow these guidelines to create effective and maintainable decorators:

- **Use functools.wraps:** Always preserve the original function’s metadata
- **Handle all arguments:** Use ***args, **kwargs** to support any function signature
- **Keep decorators focused:** Each decorator should do one thing well
- **Document decorators:** Clearly explain what your decorator does
- **Consider performance:** Decorators add overhead to function calls
- **Test decorated functions:** Ensure decorators don’t change expected behavior

9. Conclusion

Python decorators embody elegant metaprogramming by providing a clean syntax for extending function and class behavior. They allow developers to apply consistent patterns across their codebase, separate concerns, and write more maintainable software.
By mastering decorators, you can:

- Add cross-cutting functionality without cluttering core business logic
- Create reusable code patterns that can be applied consistently

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- Solve common programming challenges with clean, readable solutions
- Better understand Python's powerful metaprogramming capabilities

Decorators shine brightest when they handle aspects like logging, timing, caching, authentication, and validation—allowing your core code to focus solely on its primary responsibility.

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

Decorators: Transform Your Code

How will you leverage decorators in your next project?