Chapter 8: Classes and Decorators

Object-Oriented Programming and Function Enhancement in Python

CS 101 - Fall 2025

On For Today

Let's dive into Python's powerful organizational features!

Topics covered in today's discussion:

- **What are Classes?** - Building custom data types
- **Objects and Instances** - Creating and using objects
- **Methods and Attributes** - Adding behavior and properties
- **Constructors and Inheritance** - Advanced class concepts
- **What are Decorators?** - Function enhancers explained
- **Built-in Decorators** - `@property`, `@staticmethod`, `@classmethod`
- **Custom Decorators** - Creating your own function wrappers
- **Real-World Applications** - Classes and decorators in action

Get Ready for Classes and Decorators!

Ready to build amazing Python objects and enhance functions!

Part 1: Python Classes

> What We'll Learn
>
> Classes are blueprints for creating objects - they help us organize code and model real-world concepts!

Think of classes as: Cookie cutters that create cookie objects - same shape, different decorations!

What Are Classes?

> Definition
>
> **Classes** are templates for creating objects. They bundle data (attributes) and functions (methods) together to model real-world concepts or abstract ideas.

Think of them as: Blueprints for houses - each house (object) follows the blueprint (class) but can have different colors, sizes, and features!

Classes: The Basics

Basic Class Syntax

```
# Basic class definition
class Dog:
    # Class attribute (shared by all instances)
    species = "Canis lupus"

    # Constructor method (__init__)
    def __init__(self, name, age):
        # Instance attributes (unique to each object)
        self.name = name
        self.age = age

    # Instance method
    def bark(self):
        return f"{self.name} says Woof!"

    # Another method with parameters
    def celebrate_birthday(self):
        self.age += 1
        return f"Happy birthday {self.name}! Now {self.age} years old!"

# Creating objects (instances)
my_dog = Dog("Buddy", 3)
your_dog = Dog("Luna", 5)

print(my_dog.bark())              # Buddy says Woof!
print(your_dog.celebrate_birthday())  # Happy birthday Luna! Now 6 years old!
```

Key Class Concepts Explained

Important Terms

Class: The blueprint or template
Object/Instance: A specific creation from the class
Attributes: Variables that store data
Methods: Functions that belong to the class
`self`: Reference to the current instance
`__init__`: Constructor method that runs when creating objects

Working with Objects

Creating Class Objects

```
class Car:
    # Class attribute
    wheels = 4

    # Constructor
    def __init__(self, make, model, year):
        self.make = make      # Instance attribute
        self.model = model    # Instance attribute
        self.year = year      # Instance attribute
        self.odometer = 0     # Default instance attribute

    # Method
    def drive(self, miles):
        self.odometer += miles
        return f"Drove {miles} miles. Total: {self.odometer}"
```

Key Point: A class is used to create an object. Each instance is own object. Neat-O!

Working with Objects

Creating and Using Objects

```python
# Create instances
car1 = Car("Toyota", "Camry", 2022)
car2 = Car("Honda", "Civic", 2021)

# Access attributes
print(f"Car 1: {car1.make} {car1.model}")  # Toyota Camry
print(f"Car 2: {car2.year}")               # 2021

# Call methods
print(car1.drive(100))  # Drove 100 miles. Total: 100
print(car1.drive(50))   # Drove 50 miles. Total: 150

# Modify attributes
car2.make = "Acura"  # Changing attribute value
print(car2.make)     # Acura

# Access class attributes
print(Car.wheels)    # 4 (from class)
print(car1.wheels)   # 4 (inherited from class)
```

Key Point: Each object has its own copy of instance attributes but shares class attributes!

Quick Challenge #1 (3 minutes)

Your Turn: Basic Class Creation

Challenge: Create a `Student` class with the following features:

1. Attributes: `name`, `student_id`, `grade_level`, and `gpa` (default to 0.0)
2. Methods:
    - `introduce()` - returns "Hi, I'm [name], student ID [id]"
    - `update_gpa(new_gpa)` - updates the GPA
    - `is_honor_student()` - returns True if GPA >= 3.5

Starter Code:

```python
class Student:
    # Your code here
    pass

# Test your class
student1 = Student("Alice", "S001", 10)
print(student1.introduce())
student1.update_gpa(3.8)
print(f"Honor student: {student1.is_honor_student()}")
```

Challenge #1 Solutions

Solutions

```python
class Student:
    def __init__(self, name, student_id, grade_level, gpa=0.0):
        self.name = name
        self.student_id = student_id
        self.grade_level = grade_level
        self.gpa = gpa

    def introduce(self):
        return f"Hi, I'm {self.name}, student ID {self.student_id}"

    def update_gpa(self, new_gpa):
        self.gpa = new_gpa
        return f"GPA updated to {self.gpa}"

    def is_honor_student(self):
        return self.gpa >= 3.5

# Test results
student1 = Student("Alice", "S001", 10)
print(student1.introduce())        # Hi, I'm Alice, student ID S001
student1.update_gpa(3.8)
print(f"Honor student: {student1.is_honor_student()}")  # True

student2 = Student("Bob", "S002", 11, 3.2)
print(f"Bob honor status: {student2.is_honor_student()}")  # False
```

Class Inheritance

Inheritance Basics

Inheritance allows you to create a new class based on an existing class. The new class inherits attributes and methods from the parent class!

Think of it as: Family traits - children inherit characteristics from parents but can also have their own unique features!

Inheritance Example

Parent and Child Classes

```python
# Parent class (Base class)
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.energy = 100

    def eat(self):
        self.energy += 10
        return f"{self.name} is eating. Energy: {self.energy}"

    def sleep(self):
        self.energy += 20
        return f"{self.name} is sleeping. Energy: {self.energy}"

# Child class inherits from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Canine")  # Call parent constructor
        self.breed = breed

    def bark(self):  # New method specific to Dog
        return f"{self.name} barks! Woof woof!"

    def play_fetch(self):  # Another dog-specific method
        self.energy -= 15
        return f"{self.name} plays fetch! Energy: {self.energy}"

# Using inheritance
my_dog = Dog("Max", "Golden Retriever")
print(my_dog.eat())        # Inherited from Animal
print(my_dog.bark())       # Dog-specific method
print(my_dog.play_fetch()) # Dog-specific method
```

Method Overriding

Customizing Inherited Methods

```
class Bird(Animal):
    def __init__(self, name, can_fly=True):
        super().__init__(name, "Avian")
        self.can_fly = can_fly

    # Override the sleep method from Animal
    def sleep(self):
        self.energy += 30  # Birds need more rest!
        return f"{self.name} sleeps in a nest. Energy: {self.energy}"

    # New method
    def fly(self):
        if self.can_fly:
            self.energy -= 20
            return f"{self.name} soars through the sky!"
        else:
            return f"{self.name} cannot fly."

# Comparison
dog = Dog("Buddy", "Labrador")
bird = Bird("Tweety")

print(dog.sleep())   # Uses Animal's sleep method (energy +20)
print(bird.sleep())  # Uses Bird's overridden sleep method (energy +30)
print(bird.fly())    # Bird-specific method
```

Key Point: Child classes can override parent methods to provide specialized behavior!

Quick Challenge #2 (3 minutes)

Your Turn: Inheritance Practice

Challenge: Create a `Vehicle` parent class and a `Motorcycle` child class:
Vehicle class: - Attributes: `make`, `model`, `year`, `fuel_level` (default 100) - Methods: `start_engine()`, `refuel()`
Motorcycle class: - Inherits from Vehicle - Additional attribute: `engine_size` - New

method: `wheelie()` (reduces fuel by 5) - Override `start_engine()` to include motorcycle-specific message
Starter Code:

```
class Vehicle:
    # Your Vehicle class here
    pass

class Motorcycle(Vehicle):
    # Your Motorcycle class here
    pass

# Test your classes
bike = Motorcycle("Harley", "Sportster", 2023, 883)
print(bike.start_engine())
print(bike.wheelie())
```

Challenge #2 Solutions

Solutions

```
class Vehicle:
    def __init__(self, make, model, year, fuel_level=100):
        self.make = make
        self.model = model
        self.year = year
        self.fuel_level = fuel_level

    def start_engine(self):
        return f"{self.year} {self.make} {self.model} engine started!"

    def refuel(self):
        self.fuel_level = 100
        return f"Tank refueled to {self.fuel_level}%"

class Motorcycle(Vehicle):
    def __init__(self, make, model, year, engine_size, fuel_level=100):
        super().__init__(make, model, year, fuel_level)
        self.engine_size = engine_size

    def start_engine(self):  # Override parent method
        return f"{self.year} {self.make} {self.model} motorcycle roars to life!  "

    def wheelie(self):
        self.fuel_level -= 5
        return f"Awesome wheelie! Fuel: {self.fuel_level}%"

# Test results
bike = Motorcycle("Harley", "Sportster", 2023, 883)
print(bike.start_engine())  # Motorcycle roars to life!
print(bike.wheelie())       # Awesome wheelie! Fuel: 95%
print(bike.refuel())        # Tank refueled to 100%
```

Part 2: Python Decorators

> What We'll Learn
>
> Decorators are a way to modify or enhance functions without changing their code. They're like gift wrapping for functions!

Think of decorators as: Gift wrapping - the present (function) stays the same, but the wrapper adds something special!

What Are Decorators?

> Definition
>
> **Decorators** are functions that take another function as input and extend or modify its behavior without permanently changing it. They use the `@decorator_name` syntax!

Think of them as: Function enhancers - like adding superpowers to your existing functions!

Decorators: The Basics

Basic Decorator Syntax

```python
# Simple decorator function
def my_decorator(func):
    def wrapper():
        print("Something before the function")
        result = func()  # Call the original function
        print("Something after the function")
        return result
    return wrapper

# Using the decorator with @ syntax
@my_decorator
def say_hello():
    print("Hello, World!")

# When we call say_hello(), it's actually wrapped
say_hello()

# Output:
# Something before the function
# Hello, World!
# Something after the function
```

Key Point: The `@decorator_name` syntax is equivalent to `func = decorator_name(func)`!

Understanding How Decorators Work

Step-by-Step Breakdown

What happens when you use `@my_decorator`:

1. **Decorator function** receives the original function as input
2. **Wrapper function** is created inside the decorator
3. **Wrapper function** calls the original function and adds extra behavior
4. **Decorator returns** the wrapper function
5. **Original function name** now points to the wrapper function

Decorators with Arguments

Handling Function Parameters

```python
# This decorator adds timing to functions
def timer_decorator(func):
    def wrapper():
        import time
        start_time = time.time()
        result = func()
        end_time = time.time()
        print(f"Function took {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timer_decorator
def slow_function():
    import time
    time.sleep(1)
    return "Done!"

slow_function()  # Function took 1.0041 seconds
```

Setting Up Decorators with Arguments

Handling Function Parameters

```
# Decorator that works with functions that have arguments
def logging_decorator(func):
    def wrapper(*args, **kwargs):  # Accept any arguments
        print(f"Calling function: {func.__name__}")
        print(f"Arguments: args={args}, kwargs={kwargs}")
        result = func(*args, **kwargs)  # Pass arguments to original function
        print(f"Function returned: {result}")
        return result
    return wrapper

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

@logging_decorator
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

# Test the decorated functions
result1 = add_numbers(5, 3)
print(f"Result: {result1}")

result2 = greet("Alice", greeting="Hi")
print(f"Result: {result2}")
```

Built-in Decorators: @property, @staticmethod and @classmethod

! Important

- The `@staticmethod` decorator in Python marks a method inside a class that does not require access to the class or any instance of the class. It behaves like a regular function but is contained within the class's namespace for logical organization.
- The `@classmethod` decorator is a built-in decorator used to define a method that belongs to the class itself, rather than to an instance of the class.
- The `@property` decorator is a built-in decorator used to define managed attributes within a class. It allows you to transform regular methods into "properties," which can be accessed like attributes but have underlying getter, setter, and deleter methods to control their behavior.

Using Built-in Decorators: @property

The @property Decorator

@property turns a method into a readable attribute. It's used for computed properties and data validation!

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius  # Private attribute

    @property
    def radius(self):
        """Getter for radius"""
        return self._radius

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

    @property
    def area(self):
        """Computed property - calculated on demand"""
        import math
        return math.pi * self._radius ** 2

    @property
    def diameter(self):
        """Another computed property"""
        return 2 * self._radius

# Using the Circle class
circle = Circle(5)
print(f"Radius: {circle.radius}")     # Uses @property getter
print(f"Area: {circle.area:.2f}")     # Computed property
print(f"Diameter: {circle.diameter}") # Another computed property

circle.radius = 10  # Uses @radius.setter
print(f"New area: {circle.area:.2f}")
```

Using Built-in Decorators: @staticmethod and @classmethod

Class-related Decorators

```python
class MathUtils:
    pi = 3.14159

    @staticmethod
    def add(x, y):
        """Static method - doesn't need self or cls"""
        return x + y

    @classmethod
    def circle_area(cls, radius):
        """Class method - receives cls (the class) as first argument"""
        return cls.pi * radius ** 2

    def instance_method(self):
        """Regular instance method - needs self"""
        return "This is an instance method"

# Using different types of methods
# Static method - can call without creating instance
result = MathUtils.add(5, 3)
print(f"Static method result: {result}")

# Class method - can call without creating instance
area = MathUtils.circle_area(10)
print(f"Circle area: {area:.2f}")

# Instance method - needs an instance
utils = MathUtils()
print(utils.instance_method())

# You can also call static and class methods on instances
print(f"Instance calling static: {utils.add(2, 7)}")
print(f"Instance calling class method: {utils.circle_area(3):.2f}")
```

Quick Challenge #3 (3 minutes)

Your Turn: Creating Custom Decorators

Challenge: Create a decorator called `repeat_decorator` that runs a function multiple times:

1. The decorator should take a parameter for how many times to repeat
2. It should print the result each time
3. Return the result from the last execution

Bonus: Create a `BankAccount` class with `@property` for balance validation
Starter Code:

```python
# Create your decorator here
def repeat_decorator(times):
    # Your decorator code here
    pass

@repeat_decorator(3)
def roll_dice():
    import random
    return random.randint(1, 6)

# Test your decorator
result = roll_dice()

# Bonus: BankAccount class
class BankAccount:
    # Your class with @property here
    pass
```

Challenge #3 Solutions (I)

Solutions

```python
# Solution 1: Custom repeat decorator
def repeat_decorator(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            result = None
            for i in range(times):
                result = func(*args, **kwargs)
                print(f"Execution {i+1}: {result}")
            return result
        return wrapper
    return decorator

@repeat_decorator(3)
def roll_dice():
    import random
    return random.randint(1, 6)

print("Rolling dice 3 times:")
final_result = roll_dice()
print(f"Final result: {final_result}")
```

Challenge #3 Solutions (II)

Solutions

```python
# Solution 2: BankAccount with @property
class BankAccount:
    def __init__(self, initial_balance=0):
        self._balance = initial_balance

    @property
    def balance(self):
        return self._balance

    # The below setter provides a controlled way to modify
    # the balance with validation, but the current code
    # does not demonstrate this functionality. It is a
    # useful feature for ensuring data integrity when
    # someone directly assigns to the balance property.

    @balance.setter
    def balance(self, amount):
        if amount < 0:
            raise ValueError("Balance cannot be negative!")
        self._balance = amount

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            return f"Deposited ${amount}. New balance: ${self._balance}"

    def withdraw(self, amount):
        if amount > self._balance:
            return "Insufficient funds!"
        self._balance -= amount
        return f"Withdrew ${amount}. New balance: ${self._balance}"

# Test BankAccount
account = BankAccount(100)
print(f"Initial balance: ${account.balance}")
print(account.deposit(50))
print(account.withdraw(30))


# This would trigger the setter
account.balance = 200  # This calls the setter method

# This would trigger the validation
try:
    account.balance = -50  # This would raise ValueError
except ValueError as e:
    print(f"Error: {e}")
```

Real-World Applications

> Practical Uses
>
> Classes and decorators are everywhere in real Python applications! Let's see some practical examples you'll encounter.

Real-World Example: E-commerce System

> Note
>
> Classes for Shopping Cart

```python
class Product:
    def __init__(self, name, price, category):
        self.name = name
        self.price = price
        self.category = category

    def __str__(self):
        return f"{self.name} (${self.price:.2f})"

class ShoppingCart:
    def __init__(self):
        self.items = []
        self._discount_rate = 0.0

    def add_item(self, product, quantity=1):
        self.items.append({"product": product, "quantity": quantity})
        return f"Added {quantity}x {product.name} to cart"

    @property
    def total(self):
        subtotal = sum(item["product"].price * item["quantity"]
                      for item in self.items)
        return subtotal * (1 - self._discount_rate)

    @property
    def item_count(self):
        return sum(item["quantity"] for item in self.items)

    def apply_discount(self, rate):
        if 0 <= rate <= 1:
            self._discount_rate = rate
            return f"Applied {rate*100}% discount"
        return "Invalid discount rate"

# Using the e-commerce system
laptop = Product("Gaming Laptop", 1299.99, "Electronics")
mouse = Product("Wireless Mouse", 29.99, "Electronics")

cart = ShoppingCart()
print(cart.add_item(laptop))
print(cart.add_item(mouse, 2))

print(f"Items in cart: {cart.item_count}")
print(f"Total: ${cart.total:.2f}")

print(cart.apply_discount(0.1))  # 10% discount
print(f"Total after discount: ${cart.total:.2f}")
```

Output:

```python
class Product:
    def __init__(self, name, price, category):
        self.name = name
        self.price = price
        self.category = category

    def __str__(self):
        return f"{self.name} (${self.price:.2f})"

class ShoppingCart:
    def __init__(self):
        self.items = []
        self._discount_rate = 0.0

    def add_item(self, product, quantity=1):
        self.items.append({"product": product, "quantity": quantity})
        return f"Added {quantity}x {product.name} to cart"

    @property
    def total(self):
        subtotal = sum(item["product"].price * item["quantity"]
                       for item in self.items)
        return subtotal * (1 - self._discount_rate)

    @property
    def item_count(self):
        return sum(item["quantity"] for item in self.items)

    def apply_discount(self, rate):
        if 0 <= rate <= 1:
            self._discount_rate = rate
            return f"Applied {rate*100}% discount"
        return "Invalid discount rate"

# Using the e-commerce system
laptop = Product("Gaming Laptop", 1299.99, "Electronics")
mouse = Product("Wireless Mouse", 29.99, "Electronics")

cart = ShoppingCart()
print(cart.add_item(laptop))
print(cart.add_item(mouse, 2))

print(f"Items in cart: {cart.item_count}")
print(f"Total: ${cart.total:.2f}")

print(cart.apply_discount(0.1))  # 10% discount
print(f"Total after discount: ${cart.total:.2f}")
```

```
Added 1x Gaming Laptop to cart
Added 2x Wireless Mouse to cart
Items in cart: 3
Total: $1359.97
Applied 10.0% discount
Total after discount: $1223.97
```

Real-World Example: Game Development

> **Note**
>
> Classes with Inheritance for Game Characters

```python
class GameCharacter:
    def __init__(self, name, health=100, attack_power=10):
        self.name = name
        self.max_health = health
        self._health = health
        self.attack_power = attack_power
        self.level = 1

    @property
    def health(self):
        return self._health

    @health.setter
    def health(self, value):
        self._health = max(0, min(value, self.max_health))  # Keep between 0 and max

    @property
    def is_alive(self):
        return self._health > 0

    def attack(self, target):
        if not self.is_alive:
            return f"{self.name} cannot attack - defeated!"

        damage = self.attack_power
        target.health -= damage
        return f"{self.name} attacks {target.name} for {damage} damage!"

class Warrior(GameCharacter):
    def __init__(self, name):
        super().__init__(name, health=150, attack_power=15)
        self.armor = 5

    def shield_bash(self, target):
        damage = self.attack_power + self.armor
        target.health -= damage
        return f"{self.name} shield bashes {target.name} for {damage} damage!"

class Mage(GameCharacter):
    def __init__(self, name):
        super().__init__(name, health=80, attack_power=20)
        self.mana = 100

    def cast_fireball(self, target):
        if self.mana < 20:
            return f"{self.name} is out of mana!"

        self.mana -= 20
        damage = self.attack_power * 2
        target.health -= damage
        return f"{self.name} casts fireball on {target.name} for {damage} damage!"

# Game battle simulation
```

Output:

```python
class GameCharacter:
    def __init__(self, name, health=100, attack_power=10):
        self.name = name
        self.max_health = health
        self._health = health
        self.attack_power = attack_power
        self.level = 1

    @property
    def health(self):
        return self._health

    @health.setter
    def health(self, value):
        self._health = max(0, min(value, self.max_health))  # Keep between 0 and max

    @property
    def is_alive(self):
        return self._health > 0

    def attack(self, target):
        if not self.is_alive:
            return f"{self.name} cannot attack - defeated!"

        damage = self.attack_power
        target.health -= damage
        return f"{self.name} attacks {target.name} for {damage} damage!"

class Warrior(GameCharacter):
    def __init__(self, name):
        super().__init__(name, health=150, attack_power=15)
        self.armor = 5

    def shield_bash(self, target):
        damage = self.attack_power + self.armor
        target.health -= damage
        return f"{self.name} shield bashes {target.name} for {damage} damage!"

class Mage(GameCharacter):
    def __init__(self, name):
        super().__init__(name, health=80, attack_power=20)
        self.mana = 100

    def cast_fireball(self, target):
        if self.mana < 20:
            return f"{self.name} is out of mana!"

        self.mana -= 20
        damage = self.attack_power * 2
        target.health -= damage
        return f"{self.name} casts fireball on {target.name} for {damage} damage!"

# Game battle simulation
```

```
Warrior: Sir Lancelot (Health: 150)
Mage: Gandalf (Health: 80)

Battle begins!
Sir Lancelot attacks Gandalf for 15 damage!
Mage health: 65
Gandalf casts fireball on Sir Lancelot for 40 damage!
Warrior health: 110
Sir Lancelot shield bashes Gandalf for 20 damage!
Mage health: 45
Mage alive: True
```

Real-World Example: Web API with Decorators

> **Note**
>
> Decorators for Web Development

```
import time
from functools import wraps

# Authentication decorator
def require_auth(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Simulate checking authentication
        user_authenticated = True  # In real app, check session/token
        if not user_authenticated:
            return {"error": "Authentication required", "status": 401}
        return func(*args, **kwargs)
    return wrapper

# Rate limiting decorator
def rate_limit(max_calls=5, time_window=60):
    call_times = []

    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            now = time.time()
            # Remove old calls outside time window
            call_times[:] = [t for t in call_times if now - t < time_window]

            if len(call_times) >= max_calls:
                return {"error": "Rate limit exceeded", "status": 429}

            call_times.append(now)
            return func(*args, **kwargs)
        return wrapper
    return decorator

# Logging decorator
def log_api_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"API call: {func.__name__} at {time.strftime('%Y-%m-%d %H:%M:%S')}")
        result = func(*args, **kwargs)
        print(f"API response: {result}")
        return result
    return wrapper

# API endpoint class
class UserAPI:
    def __init__(self):
        self.users = {
            1: {"name": "Alice", "email": "alice@email.com"},
            2: {"name": "Bob", "email": "bob@email.com"}
        }

    @log_api_call
    @require_auth
```

Output:

```python
import time
from functools import wraps

# Authentication decorator
def require_auth(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Simulate checking authentication
        user_authenticated = True  # In real app, check session/token
        if not user_authenticated:
            return {"error": "Authentication required", "status": 401}
        return func(*args, **kwargs)
    return wrapper

# Rate limiting decorator
def rate_limit(max_calls=5, time_window=60):
    call_times = []

    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            now = time.time()
            # Remove old calls outside time window
            call_times[:] = [t for t in call_times if now - t < time_window]

            if len(call_times) >= max_calls:
                return {"error": "Rate limit exceeded", "status": 429}

            call_times.append(now)
            return func(*args, **kwargs)
        return wrapper
    return decorator

# Logging decorator
def log_api_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"API call: {func.__name__} at {time.strftime('%Y-%m-%d %H:%M:%S')}")
        result = func(*args, **kwargs)
        print(f"API response: {result}")
        return result
    return wrapper

# API endpoint class
class UserAPI:
    def __init__(self):
        self.users = {
            1: {"name": "Alice", "email": "alice@email.com"},
            2: {"name": "Bob", "email": "bob@email.com"}
        }

    @log_api_call
    @require_auth
```

```
=== API Testing ===
API call: get_user at 2025-11-13 12:21:28
API response: {'user': {'name': 'Alice', 'email': 'alice@email.com'}, 'status': 200}
{'user': {'name': 'Alice', 'email': 'alice@email.com'}, 'status': 200}
API call: get_user at 2025-11-13 12:21:28
API response: {'user': {'name': 'Bob', 'email': 'bob@email.com'}, 'status': 200}
{'user': {'name': 'Bob', 'email': 'bob@email.com'}, 'status': 200}
API call: create_user at 2025-11-13 12:21:28
API response: {'user': {'name': 'Charlie', 'email': 'charlie@email.com'}, 'id': 3, 'status
{'user': {'name': 'Charlie', 'email': 'charlie@email.com'}, 'id': 3, 'status': 201}
```

Final Challenge: Putting It All Together (5 minutes)

Your Turn: Library Management System

Challenge: Create a complete library system using classes, inheritance, and decorators:
Requirements: 1. `Book` class with title, author, ISBN, and available status 2. `Library` class that manages books with methods to add/checkout/return books 3. `Member` class that inherits from a `Person` base class 4. Use `@property` for computed attributes 5. Create a decorator that logs all library operations
Bonus: Add different types of books (TextBook, Novel) with specific behaviors
Starter Code:

```python
# Create your classes and decorators here

# Test your system
library = Library("City Library")
book1 = Book("Python Programming", "John Doe", "123456789", True)
member1 = Member("Alice", "M001", "alice@email.com")

# Test the system
library.add_book(book1)
library.checkout_book("123456789", member1)
```

Final Challenge: Complete Solution

Library Management System Solution

```
from datetime import datetime
from functools import wraps

# Logging decorator for library operations
def log_operation(func):
```

Best Practices (I)

Writing Clean Object-Oriented Code

Classes Best Practices: * Use clear, descriptive class names (PascalCase) * Keep classes focused on a single responsibility * Use `@property` for computed attributes and validation * Document your classes and methods * Use inheritance when there's a clear "is-a" relationship

Decorators Best Practices: * Use `@wraps` from `functools` to preserve function metadata * Keep decorators simple and focused * Document what your decorators do * Consider performance impact of decorators * Use built-in decorators when appropriate

Best Practices (II)

Writing Clean Object-Oriented Code

General Guidelines: * Favor composition over inheritance when relationships are complex * Use descriptive variable and method names * Keep methods short and focused * Test your classes and decorators thoroughly

Summary (I)

What You've Learned Today

Classes Mastery: * Creating classes with attributes and methods * Understanding constructors (`__init__`) and `self` * Implementing inheritance and method overriding * Using `@property` for computed attributes and validation

Decorators Mastery: * Understanding how decorators modify function behavior * Creating custom decorators with and without parameters * Using built-in decorators: `@property`, `@staticmethod`, `@classmethod` * Applying decorators in real-world scenarios

Summary (II)

What You've Learned Today

Real-World Applications: * E-commerce systems with product and cart classes * Game development with character inheritance * Web APIs with authentication and logging decorators * Library management systems combining classes and decorators

Congrats!

Next Steps: Continue Your Journey

Where to Go From Here

Immediate Practice: * Create classes for your own projects * Experiment with multiple inheritance * Build custom decorators for common patterns * Practice with `@dataclass` decorator
Advanced Topics to Explore: * Abstract Base Classes (ABC) * Context managers (`__enter__`, `__exit__`) * Metaclasses and advanced decorators * Design patterns (Singleton, Factory, Observer)
Resources: * "Effective Python" by Brett Slatkin * Python's `abc` and `dataclasses` modules * Practice building larger object-oriented projects * Explore web frameworks like Flask/Django that use decorators extensively