🎯 Ultimate Deep Dive: Object-Oriented Programming (OOP) in Python

1. The Simple Explanation (The 'Feynman' Analogy) 🧠

OOP matlab **real-world cheezein code mein represent karna**. Dekho, agar tumhe ek car banana hai code mein, toh traditional programming mein tum alag-alag variables aur functions banate: `car_color` , `car_speed` , `drive_car()` , etc. Lekin OOP mein, tum ek **blueprint** (Class) banate ho jisme saari properties aur behaviors ek saath hote hain.

Class = Blueprint/Template (jaise architectural drawing)
Object = Us blueprint se bani actual cheez (jaise us drawing se bana actual ghar)

```python
# Yeh hai ek Class - Car ka blueprint
class Car:
    def __init__(self, color, brand):
        self.color = color      # Instance variable - har car ka apna color
        self.brand = brand      # Instance variable - har car ka apna brand

    def drive(self):            # Method - behavior
        print(f"{self.brand} car chal rahi hai!")

# Ab objects banana - blueprint se actual cars
my_car = Car("Red", "BMW")      # Object 1
your_car = Car("Blue", "Audi")   # Object 2
```

4 Pillars of OOP:

1. **Encapsulation** 📦 - Data aur methods ko ek hi unit mein pack karna. Private/Public access control.
2. **Inheritance** 🧬 - Ek class dusri class ki properties inherit kar sakti hai (parent-child relationship)
3. **Polymorphism** 🎭 - Same naam ke methods different tareeke se kaam kar sakte hain
4. **Abstraction** 🎨 - Complex implementation ko hide karke sirf essential features dikhana

2. Intuitive Analogies & Real-Life Examples 🌟

Analogy 1: Restaurant Kitchen 🍳

```python
class Chef:  # Blueprint for all chefs
    def __init__(self, name, specialty):
        self.name = name
        self.specialty = specialty

    def cook(self, dish):
        print(f"{self.name} is cooking {dish}")

# Different chefs (objects) same blueprint se bane
italian_chef = Chef("Mario", "Pasta")
indian_chef = Chef("Rajesh", "Biryani")
```

Restaurant mein har chef ka apna specialization hai, lekin sabke paas basic cooking skills hain. Class = "Chef" ka concept, Objects = Individual chefs.

Analogy 2: Factory Assembly Line 🏭

Inheritance ko samjho factory ke through:

```python
class Vehicle:  # Parent/Base class
    def __init__(self, wheels):
        self.wheels = wheels

    def move(self):
        return "Moving..."

class Car(Vehicle):  # Child inherits from Vehicle
    def __init__(self, wheels, doors):
        super().__init__(wheels)  # Parent ka __init__ call karo
        self.doors = doors

    def honk(self):  # Car-specific method
        return "Beep beep!"

class Bike(Vehicle):  # Another child
    def __init__(self, wheels, has_carrier):
        super().__init__(wheels)
        self.has_carrier = has_carrier
```

Jaise factory mein basic vehicle platform se alag-alag products bante hain (car, bike, truck), waise hi base class se specialized classes banti hain.

Analogy 3: Social Media Profile System 👤

```python
class User:
    total_users = 0  # Class variable - sabke liye common

    def __init__(self, username, email):
        self.username = username  # Instance variable
        self.email = email
        self._password = None  # Private variable (convention)
        User.total_users += 1

    def set_password(self, pwd):
        self._password = self._hash_password(pwd)

    def _hash_password(self, pwd):  # Private method
        return f"hashed_{pwd}"

    @classmethod
    def get_total_users(cls):  # Class method
        return cls.total_users

    @staticmethod
    def validate_email(email):  # Static method
        return "@" in email
```

Har user ka apna profile (instance) hai, lekin total users count sabke liye common hai (class variable).

3. The Expert Mindset: How Professionals Think 🎯

Mental Models of OOP Experts:

1. "Nouns as Classes, Verbs as Methods"

```python
# Nouns = Classes
class BankAccount:
    # Verbs = Methods
    def deposit(self):
        pass

    def withdraw(self):
        pass
```

Experts pehle domain ko analyze karte hain: "System mein kaun kaun se entities hain (nouns)? Unke kya actions hain (verbs)?"

2. "Has-A vs Is-A Relationship"

```python
# IS-A relationship (Inheritance)
class Animal:
    pass

class Dog(Animal):  # Dog IS-A Animal
    pass

# HAS-A relationship (Composition)
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()  # Car HAS-A Engine
```

3. Design Process Step-by-Step:

```
Step 1: Identify Entities (Classes)
↓
Step 2: Define Attributes (Instance Variables)
↓
Step 3: Define Behaviors (Methods)
↓
Step 4: Establish Relationships (Inheritance/Composition)
↓
Step 5: Apply SOLID Principles
↓
Step 6: Refactor & Optimize
```

4. Questions Experts Ask:

- "Kya yeh entity independent hai ya kisi aur entity ka part hai?"
- "Kya inheritance zaroori hai ya composition better hoga?"
- "Kaunse data ko private rakhna chahiye?"
- "Kya yeh class ek hi responsibility handle kar rahi hai?" (Single Responsibility Principle)

4. Common Mistakes & "Pitfall Patrol" ⚠️

Mistake 1: Mutable Default Arguments 💣

```python
# ❌ WRONG - Dangerous!
class Student:
    def __init__(self, name, subjects=[]):
        self.name = name
        self.subjects = subjects  # Same list sabke liye!

s1 = Student("Raj")
s1.subjects.append("Math")
s2 = Student("Priya")
print(s2.subjects)  # Output: ['Math'] - WTF?

# ✅ CORRECT
class Student:
    def __init__(self, name, subjects=None):
        self.name = name
        self.subjects = subjects if subjects is not None else []
```

Why trap hai: Python mein default arguments sirf ek baar evaluate hote hain (function definition time pe). Mutable objects (list, dict) share ho jaate hain.

Mistake 2: `__init__` Ko Regular Method Samajhna 🤦 {#mistake-2-init-ko-regular-method-samajhna- }

```python
# ❌ WRONG
class Car:
    def __init__(self):
        return "Car created"  # __init__ kuch return nahi karta!

# ✅ CORRECT
class Car:
    def __init__(self):
        self.color = "red"
        # No return statement needed
```

Why trap hai: `__init__` constructor nahi hai, initializer hai. Yeh object ko setup karta hai, create nahi. `__new__` actual constructor hai.

Mistake 3: Class Variables vs Instance Variables Confusion 😵

```
# ❌ Dangerous Pattern
class Game:
    score = 0  # Class variable

    def add_points(self, points):
        self.score += points  # Yeh instance variable ban jata hai!

game1 = Game()
game1.add_points(10)
game2 = Game()
print(game2.score)  # Output: 0 (not 10)

# ✅ CORRECT for class variable
class Game:
    total_games = 0

    def __init__(self):
        self.score = 0  # Instance variable
        Game.total_games += 1
```

Why trap hai: Assignment (`self.score = ...`) naya instance variable create karta hai instead of class variable ko modify karne ke.

Mistake 4: Super() Ko Forget Karna (Multiple Inheritance Mein)

```
# ❌ WRONG
class A:
    def __init__(self):
        print("A init")

class B:
    def __init__(self):
        print("B init")

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)  # Directly call - MRO ko ignore karta hai

# ✅ CORRECT - Use super()
class C(A, B):
    def __init__(self):
        super().__init__()  # MRO follow karega
```

Why trap hai: Multiple inheritance mein Method Resolution Order (MRO) important hai. `super()` cooperatively chain ko follow karta hai.

Mistake 5: Properties Ko Ignore Karna 🎭

```python
# ❌ Ugly and Unpythonic
class Circle:
    def __init__(self, radius):
        self._radius = radius

    def get_radius(self):
        return self._radius

    def set_radius(self, value):
        if value < 0:
            raise ValueError("Radius negative nahi ho sakta")
        self._radius = value

c = Circle(5)
c.set_radius(10)  # Java-style getter/setter

# ✅ PYTHONIC - Use @property
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius negative nahi ho sakta")
        self._radius = value

c = Circle(5)
c.radius = 10  # Clean syntax!
```

5. Thinking Like an Architect (The 30,000-Foot View) 🏛️

System-Level Design with OOP:

SOLID Principles (The Architect's Bible):

S - Single Responsibility Principle

```python
# ❌ BAD - Too many responsibilities
class User:
    def __init__(self, name):
        self.name = name

    def save_to_db(self):
        # Database logic
        pass

    def send_email(self):
        # Email logic
        pass

# ✅ GOOD - Separate responsibilities
class User:
    def __init__(self, name):
        self.name = name

class UserRepository:
    def save(self, user):
        # Database logic
        pass

class EmailService:
    def send(self, user):
        # Email logic
        pass
```

O - Open/Closed Principle

```python
from abc import ABC, abstractmethod

# Open for extension, closed for modification
class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

class CreditCardProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing ${amount} via Credit Card")

class PayPalProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing ${amount} via PayPal")

# New payment method? Extend, don't modify!
class CryptoProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing ${amount} via Crypto")
```

L - Liskov Substitution Principle

```python
# Child class parent ki jagah kaam kar sakti hai
class Bird:
    def fly(self):
        return "Flying"

class Sparrow(Bird):
    pass  # Can fly, follows LSP

# ❌ VIOLATES LSP
class Penguin(Bird):
    def fly(self):
        raise Exception("Penguins can't fly!")  # Breaks contract
```

I - Interface Segregation

```python
# ❌ BAD - Fat interface
class Worker(ABC):
    @abstractmethod
    def work(self):
        pass

    @abstractmethod
    def eat(self):
        pass

class Robot(Worker):  # Robot eat nahi karta!
    def eat(self):
        pass  # Forced to implement

# ✅ GOOD - Segregated interfaces
class Workable(ABC):
    @abstractmethod
    def work(self):
        pass

class Eatable(ABC):
    @abstractmethod
    def eat(self):
        pass

class Human(Workable, Eatable):
    def work(self):
        pass

    def eat(self):
        pass

class Robot(Workable):  # Only what's needed
    def work(self):
        pass
```

D - Dependency Inversion

```
# ❌ BAD - High-level depends on low-level
class MySQLDatabase:
    def save(self, data):
        print("Saving to MySQL")

class UserService:
    def __init__(self):
        self.db = MySQLDatabase()  # Tightly coupled

# ✅ GOOD - Depend on abstractions
class Database(ABC):
    @abstractmethod
    def save(self, data):
        pass

class MySQLDatabase(Database):
    def save(self, data):
        print("Saving to MySQL")

class PostgreSQLDatabase(Database):
    def save(self, data):
        print("Saving to PostgreSQL")

class UserService:
    def __init__(self, db: Database):  # Depends on abstraction
        self.db = db
```

Key Architectural Trade-offs:

Inheritance vs Composition:

```
# Inheritance - IS-A
class Employee:
    pass

class Manager(Employee):  # Manager IS-A Employee
    pass

# Composition - HAS-A (Generally preferred)
class Engine:
    pass

class Car:
    def __init__(self):
        self.engine = Engine()  # Car HAS-A Engine
```

Trade-off: Composition flexible hai, but inheritance code reuse easy hai. Mantra: **"Favor composition over inheritance"**

6. Real-World Applications (Where It's Hiding in Plain Sight) 🌍

Application 1: Django Web Framework 🌐

```
from django.db import models

class BlogPost(models.Model):  # Inheritance from Model
    title = models.CharField(max_length=200)
    content = models.TextField()
    published_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):  # Magic method
        return self.title

    class Meta:  # Nested class for metadata
        ordering = ['-published_date']
```

How: Django uses OOP heavily - Models (classes) represent database tables, Views (classes) handle requests, Forms (classes) validate data. Har component ek class hai jo specific responsibility handle karta hai.

Application 2: Game Development (Pygame/Unity) 🎮

```python
class GameObject:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def update(self):
        pass

    def render(self):
        pass

class Player(GameObject):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.health = 100
        self.speed = 5

    def move(self, dx, dy):
        self.x += dx * self.speed
        self.y += dy * self.speed

class Enemy(GameObject):
    def __init__(self, x, y, ai_type):
        super().__init__(x, y)
        self.ai = ai_type

    def attack(self, target):
        target.health -= 10
```

How: Har game entity (player, enemy, item) ek object hai. Inheritance se common behaviors share hote hain, polymorphism se different entities apne tareeke se behave karte hain.

Application 3: Scikit-Learn (Machine Learning) 🤖

```
from sklearn.base import BaseEstimator, ClassifierMixin

class CustomClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, param1=1.0):
        self.param1 = param1

    def fit(self, X, y):
        # Training logic
        return self

    def predict(self, X):
        # Prediction logic
        return predictions
```

How: Sklearn ka pura architecture OOP pe based hai. Har algorithm ek class hai jo common interface (`fit` , `predict`) follow karta hai. Polymorphism se tum kisi bhi model ko swap kar sakte ho.

Application 4: FastAPI (Your NEETPrepGPT Backend) ⚡

```python
from fastapi import FastAPI
from pydantic import BaseModel

class MCQRequest(BaseModel):  # Pydantic uses OOP for validation
    topic: str
    difficulty: str
    count: int

class MCQResponse(BaseModel):
    question: str
    options: list
    correct_answer: str

app = FastAPI()

class MCQGenerator:
    def __init__(self, openai_client):
        self.client = openai_client

    def generate(self, request: MCQRequest) -> list[MCQResponse]:
        # Generation logic
        pass

# Dependency Injection (OOP pattern)
generator = MCQGenerator(openai_client)

@app.post("/generate")
def generate_mcqs(request: MCQRequest):
    return generator.generate(request)
```

How: FastAPI internally OOP use karta hai for request validation (Pydantic models), routing, dependency injection. Tumhara entire backend OOP principles pe chalega.

Application 5: Selenium WebDriver (Web Scraping) 🕷️

```python
from selenium import webdriver
from selenium.webdriver.common.by import By

class NEETScraper:
    def __init__(self, url):
        self.driver = webdriver.Chrome()  # Object
        self.url = url

    def scrape_questions(self):
        self.driver.get(self.url)
        questions = self.driver.find_elements(By.CLASS_NAME, "question")
        return [q.text for q in questions]

    def __enter__(self):  # Context manager magic methods
        return self

    def __exit__(self, *args):
        self.driver.quit()

# Usage with context manager
with NEETScraper("https://example.com") as scraper:
    data = scraper.scrape_questions()
```

How: Selenium pure OOP framework hai. WebDriver, WebElement sab classes hain. Methods chain karke complex interactions handle karte hain.

7. The CTO's Strategic View (The "So What?" for Business) 💼

Why CTOs Care About OOP:

1. Code Maintainability = Lower TCO (Total Cost of Ownership)

```python
# Without OOP - Nightmare for scaling
users = []
user_emails = []
user_passwords = []

def add_user(name, email, pwd):
    users.append(name)
    user_emails.append(email)
    user_passwords.append(pwd)

# With OOP - Clean & Maintainable
class User:
    def __init__(self, name, email, pwd):
        self.name = name
        self.email = email
        self._password = self._hash(pwd)
```

Business Impact: Teams 40-60% faster debug kar sakte hain OOP code ko. Onboarding time 50% reduce.

2. Team Scalability:

OOP allows parallel development:

- Dev A works on `UserAuthentication` class
- Dev B works on `PaymentProcessor` class
- Dev C works on `NotificationService` class

No conflicts kyunki clear boundaries hain. **Result:** 3x faster feature delivery.

3. Technology Stack Evaluation Matrix:

Factor	Score (1-10)	Reasoning
Learning Curve	7	Moderate - Concepts take time
Code Reusability	9	Inheritance & Composition
Testing Ease	9	Unit tests per class easy
Scalability	8	Good for large systems
Performance	7	Slight overhead vs procedural

4. Implementation Considerations:

```python
# For NEETPrepGPT - Strategic OOP Design
class MCQEngine:
    """Core domain logic - business-critical"""
    def generate_mcq(self):
        pass

class CacheManager:
    """Performance optimization"""
    def __init__(self, redis_client):
        self.redis = redis_client

class AnalyticsTracker:
    """Business intelligence"""
    def track_user_activity(self):
        pass
```

CTO Decision Framework:

- **Small projects (<5k LOC):** OOP optional
- **Medium (5k-50k LOC):** OOP recommended
- **Large (>50k LOC):** OOP mandatory
- **Team >5 developers:** OOP essential for coordination

8. The Future of OOP (What's Next?) 🚀

Trend 1: OOP + Functional Programming Hybrid 🔄

```
from dataclasses import dataclass
from typing import List

@dataclass(frozen=True)  # Immutable OOP
class User:
    name: str
    email: str

# Functional operations on objects
users = [User("Raj", "raj@email.com"), User("Priya", "priya@email.com")]
emails = list(map(lambda u: u.email, users))  # Functional
```

Future: Languages mixing OOP's organization with FP's immutability. Python 3.10+ `dataclasses` , `match` statements show this trend.

Trend 2: Protocol-Oriented Programming (Structural Typing) 🎯

```
from typing import Protocol

class Drawable(Protocol):
    def draw(self) -> None:
        ...

class Circle:
    def draw(self) -> None:
        print("Drawing circle")

class Square:
    def draw(self) -> None:
        print("Drawing square")

def render(shape: Drawable):  # Duck typing with type hints
    shape.draw()
```

Future: Python moving towards structural subtyping (like Go interfaces). Less inheritance, more protocols.

Trend 3: AI-Generated OOP Code 🤖

```python
# AI will auto-generate boilerplate
# You: "Create a User class with email validation"
# AI generates:
class User:
    def __init__(self, name: str, email: str):
        if not self._validate_email(email):
            raise ValueError("Invalid email")
        self.name = name
        self.email = email

    @staticmethod
    def _validate_email(email: str) -> bool:
        import re
        pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
        return bool(re.match(pattern, email))
```

Impact: Developers will focus on architecture, AI will write implementation. OOP design skills become MORE valuable.

Trend 4: Type-Heavy OOP (Static Analysis) 📊

```python
from typing import TypeVar, Generic

T = TypeVar('T')

class Repository(Generic[T]):
    def __init__(self, model_class: type[T]):
        self.model = model_class

    def get_all(self) -> list[T]:
        pass

user_repo = Repository[User](User)
posts = user_repo.get_all()  # Type checker knows it's list[User]
```

Future: Python becoming more statically typed. Tools like `mypy` , `pyright` mandatory in production. OOP + strong typing = fewer bugs.

Trend 5: Meta-Programming & Decorators Evolution 🎨

```
from functools import wraps
import time

def track_performance(cls):
    """Decorator to auto-track method performance"""
    for name, method in cls.__dict__.items():
        if callable(method):
            setattr(cls, name, _time_wrapper(method))
    return cls

def _time_wrapper(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time() - start}s")
        return result
    return wrapper

@track_performance
class DataProcessor:
    def process(self):
        # Auto-tracked!
        pass
```

Future: More meta-programming for cross-cutting concerns (logging, monitoring, caching). OOP classes become "smart" with decorators.

9. AI-Powered Acceleration (Your "Unfair Advantage") 🎯

Specific Prompts for Learning OOP:

1. Class Design Prompt:

```
"Design a Python class for [your domain entity, e.g., 'Student enrollment system'].
Include: attributes, methods, inheritance structure, and follow SOLID principles.
Explain design decisions."
```

2. Code Review Prompt:

```
"Review this OOP code for:
- SOLID principle violations
- Common anti-patterns
- Performance issues
- Suggest refactoring
[Paste your code]"
```

3. Debugging Prompt:

```
"Debug this OOP code. Explain:
- Why the error is happening
- The underlying OOP concept being violated
- Step-by-step fix
[Paste error + code]"
```

4. Design Pattern Prompt:

```
"Suggest appropriate design pattern for: [your problem]
Provide:
- Pattern name
- Why it fits
- Python implementation
- Trade-offs"
```

AI-Automated OOP Tasks:

Auto-generate Boilerplate:

```
Prompt: "Generate a SQLAlchemy User model with:
- id, username, email, created_at fields
- email validation property
- password hashing
- __repr__ method"
```

Auto-write Tests:

```
Prompt: "Write pytest tests for this User class:
[paste class]
Include: test fixtures, edge cases, mock dependencies"
```

Architecture Diagram:

```
Prompt: "Create a Mermaid diagram showing class relationships for:
[describe your system]"
```

Practice with AI:

```
"Give me 5 OOP coding challenges of increasing difficulty.
For each:
- Problem statement
- Expected classes/methods
- Test cases
- Hints (don't give solution)"
```

Then solve kar and:

```
"Review my solution:
[paste code]
Compare with best practices. Rate 1-10 and suggest improvements."
```

10. Deep Thinking Triggers 🧩

1. The Ship of Theseus Problem:

```
class Ship:
    def __init__(self, parts):
        self.parts = parts

    def replace_part(self, old, new):
        self.parts.remove(old)
        self.parts.add(new)

ship = Ship({"plank1", "plank2"})
ship.replace_part("plank1", "plank3")
# Is it the same object? Same ship philosophically?
```

Think: When does an object's identity change? What is the relationship between instance identity (`id()`) and semantic identity?

2. The God Object Anti-Pattern:

Agar ek class sab kuch kar sakti hai, toh kya woh powerful hai ya poorly designed? When does a class become "too smart"?

3. Multiple Inheritance Diamond Problem:

```
class A:
    def method(self):
        print("A")

class B(A):
    def method(self):
        print("B")

class C(A):
    def method(self):
        print("C")

class D(B, C):
    pass

D().method()  # Output kya hoga? Why?
```

Think: How does Python's MRO (Method Resolution Order) solve ambiguity? Can you design a system without multiple inheritance?

4. Immutability vs Performance:

```
@dataclass(frozen=True)
class ImmutableUser:
    name: str
    email: str

# vs

class MutableUser:
    def __init__(self, name, email):
        self.name = name
        self.email = email
```

Think: When would you sacrifice performance for immutability? What are the hidden costs of mutable objects in concurrent systems?

5. Is Everything an Object?

```
print(type(5))  # <class 'int'>
print(type(int))  # <class 'type'>
print(type(type))  # <class 'type'>
```

Think: Python mein functions, classes, modules sab objects hain. What does this say about Python's meta-object protocol? How deep does the rabbit hole go?

6. Composition Recursion:

```
class Employee:
    def __init__(self, name):
        self.name = name
        self.subordinates = []

    def add_subordinate(self, emp):
        self.subordinates.append(emp)
```

Think: A manager is an employee with subordinates who are also employees. How do you model hierarchical structures? When does recursion become a problem?

7. Interfaces Without Interfaces:

Python mein formal interfaces nahi hain (unlike Java). Duck typing hai: "If it walks like a duck and

quacks like a duck, it's a duck."
Think: Is this flexibility a strength or weakness? How do you ensure contract compliance without compile-time checking?

11. Quick-Reference Cheatsheet 📋

Concept / Term	Key Takeaway / Definition
Class	Blueprint for creating objects. `class MyClass:`
Object	Instance of a class. `obj = MyClass()`
`__init__`	Initializer method (NOT constructor). Sets up object state.
`self`	Reference to the current instance. Must be first parameter in instance methods.
Instance Variable	Unique to each object. `self.variable = value`
Class Variable	Shared across all instances. Defined at class level.
Method	Function defined inside a class. Operates on object data.
Inheritance	Child class inherits parent's attributes/methods. `class Child(Parent):`
`super()`	Access parent class methods. Use in `__init__` for proper initialization.
Encapsulation	Bundle data & methods together. Use `_private` convention for internal attributes.
Polymorphism	Same interface, different implementations. Method overriding in child classes.
Abstraction	Hide complexity, show only essentials. Use ABC (Abstract Base Class).
`@property`	Pythonic way to create getters/setters. Makes methods accessible as attributes.
`@classmethod`	Method that receives class (not instance) as first arg. Use `cls` parameter.
`@staticmethod`	Method that doesn't access instance or class. Utility function inside class.
Magic Methods	`__init__` , `__str__` , `__repr__` , `__len__` , etc. Customize object behavior.

Concept / Term	Key Takeaway / Definition
MRO	Method Resolution Order. Use `ClassName.__mro__` to see lookup chain.
Composition	"HAS-A" relationship. Prefer over inheritance when possible.
Duck Typing	"If it quacks like a duck..." Type determined by methods, not class.
Mixin	Small class providing specific functionality. Multiple inheritance pattern.
SOLID	Five design principles: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.
Private Naming	`_single` = internal use (convention). `__double` = name mangling (stronger private).
`__call__`	Makes object callable like a function. `obj()` possible.
Descriptor	Objects defining `__get__` , `__set__` , `__delete__` . How `@property` works internally.
Metaclass	Class of a class. `type` is Python's default metaclass. Advanced topic.
Dataclass	`@dataclass` decorator auto-generates boilerplate (`__init__` , `__repr__` , etc).
Protocol	Structural subtyping (duck typing with type hints). PEP 544.
Avoid:	Mutable default args, god objects, deep inheritance trees (>3 levels), tight coupling

Final Pro Tip: 🔥

OOP master banne ke liye: **Read others' code** (Django, Flask, FastAPI source code), **refactor your old code** (bad OOP → good OOP), aur **think in objects** (real-world problems ko classes mein decompose karo). OOP mindset develop karne mein time lagta hai, but once it clicks, tumhara code architecture next-level ho jayega!