# 🎯 Object-Oriented Programming (OOP) - Day 16 Deep Dive

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

**What is a Class?**

A class is a blueprint or template that defines how to create objects. Think of it as a cookie cutter - it's not the cookie itself, but it defines the shape and structure of cookies you'll make.

**The `class` keyword:**

```
class MedicalTest:
    pass
```

This line says "I'm creating a blueprint called MedicalTest." The word `class` tells Python you're defining a new type of thing. `MedicalTest` is the name you're giving it. `pass` means "do nothing for now" - it's a placeholder.

**The `__init__()` Constructor:**

```
def __init__(self, test_name, result):
    self.test_name = test_name
    self.result = result
```

The `__init__()` method is a special function that runs automatically the moment you create a new object from your class. It's called a constructor because it constructs (sets up) the initial state of your object. The double underscores `__` make it "magic" - Python knows to call it automatically.

**The `self` keyword:**

`self` represents the specific object being created. When you create multiple objects from the same class, `self` helps Python know which object's data it's working with. It's like saying "this particular medical test" vs "that other medical test."

**Attributes (Instance Variables):**

```
self.test_name = test_name
self.result = result
```

These are variables that belong to the object. `self.test_name` means "store this test_name value inside this specific object." Every object gets its own copy of these variables.

**Methods (Instance Functions):**

```
def display_result(self):
    print(f"Test: {self.test_name}, Result: {self.result}")
```

Methods are functions that belong to a class and can use the object's data. They always have `self` as the first parameter so they can access the object's attributes.

**The `None` keyword:**

`None` is Python's way of saying "nothing" or "no value." It's like an empty box - there's a box, but nothing inside it.

```
self.notes = None  # This test has no notes yet
```

**Creating Objects (Instantiation):**

```
blood_test = MedicalTest("Blood Test", "Normal")
```

This creates an actual object from the blueprint. `blood_test` is now a real medical test with specific data. This process is called instantiation - you're creating an instance of the class.

# 2. Intuitive Analogies & Real-Life Examples

## Analogy 1: The House Blueprint

A **class** is like an architect's blueprint for a house. The blueprint itself isn't a house - it's the instructions for building one. When a construction company uses that blueprint to build an actual house, that house is an **object** (an instance of the blueprint).

- The blueprint defines rooms, doors, windows (like attributes)
- The blueprint defines actions like "turn on lights" or "open door" (like methods)

- Each house built from the blueprint has its own address, color, and furniture (like unique attribute values)
- You can build 100 houses from one blueprint, and each is separate

## Analogy 2: The Medical Form Template

Think of a class as a blank medical form at a hospital. The form has labeled fields: Patient Name, Age, Blood Type, Test Results. This empty form is your **class**.

When a nurse fills out the form for you specifically with your details, that completed form becomes an **object**. Your friend gets a different form filled out with their details - that's another object. Same template (class), different data (objects).

The `__init__()` method is like the nurse's action of "filling out the form the moment they pick it up." It ensures every new form gets the essential information right away.

## Analogy 3: The Digital Lego Factory

A class is like a Lego brick mold in a factory. The mold defines the shape, color options, and how pieces connect. When plastic is injected into the mold, you get an actual Lego brick (object). Each brick created is separate - you can have red ones, blue ones, big ones, small ones - all from molds (classes) that define their structure.

# 3. The Expert Mindset: How Professionals Think

## How Experts Think About OOP

Professional developers view classes through several mental models:

**Encapsulation Mindset:** "What data belongs together?" Experts group related information. If you're modeling a patient, you'd bundle name, age, medical history, and current medications into one `Patient` class rather than having scattered variables.

**Abstraction Mindset:** "What does the user of this class need to know?" Experts hide complexity. You don't need to know how a car engine works internally to drive - you just need steering, gas, and brakes. Similarly, a well-designed class exposes only essential methods.

**Reusability Mindset:** "Will I need to create many of these?" If you need to track 1000 patients, you write one `Patient` class and create 1000 objects from it. You don't write 1000 separate pieces of code.

**State vs Behavior Mindset:** Experts distinguish between:

- **State** (what an object knows): stored in attributes
- **Behavior** (what an object does): implemented in methods

# The Expert's Design Process

When starting a new project with OOP, experts follow this thought process:

#### Step 1: Identify the "Things" (Nouns)

"What are the key entities in my system?" In a medical app: Patient, Doctor, Appointment, MedicalTest, Prescription. Each becomes a potential class.

#### Step 2: Define the Data (What does each thing know?)

For a `MedicalTest` class:

- What test is this? (test_name)
- What's the result? (result)
- When was it taken? (date)
- Who ordered it? (doctor_id)

These become attributes.

#### Step 3: Define the Actions (What can each thing do?)

For a `MedicalTest` class:

- Can it display results? → `display_result()` method
- Can it check if results are abnormal? → `is_abnormal()` method
- Can it generate a report? → `generate_report()` method

#### Step 4: Determine Initialization (What's essential from the start?)

What must you know the moment you create a medical test? At minimum: test name and result. These go in `__init__()`.

#### Step 5: Ask "What Questions First?"

Before coding, experts ask:

- Will I need multiple instances of this? (If yes → class; if no → maybe just functions)
- What should be private vs public?

- How will these objects interact with each other?
- What should happen if bad data is passed?

# 4. Common Mistakes & "Pitfall Patrol"

## Mistake 1: Forgetting `self` in Method Definitions

**The Trap:**

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

    def greet():  # ❌ WRONG - Missing self
        print(f"Hello, {self.name}")
```

**Why It's a Trap:** Python automatically passes the object as the first argument when you call a method. If your method doesn't accept `self`, Python has nowhere to put that argument, causing a `TypeError`.

**How to Avoid:**

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

    def greet(self):  # ✅ CORRECT - Has self
        print(f"Hello, {self.name}")
```

**Rule:** Every instance method MUST have `self` as its first parameter, even if you don't use it.

## Mistake 2: Confusing Class Variables and Instance Variables

**The Trap:**

```
class MedicalTest:
    result = "Pending"  # ❌ TRAP - Class variable (shared by all)

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

test1 = MedicalTest("Blood Test")
test2 = MedicalTest("X-Ray")
test1.result = "Normal"

print(test2.result)  # Still prints "Pending" - confusing!
```

**Why It's a Trap:** Variables defined directly in the class (not in `__init__`) are shared among ALL objects. Changing it in one place can affect others unexpectedly.

**How to Avoid:**

```
class MedicalTest:
    def __init__(self, test_name):
        self.test_name = test_name
        self.result = "Pending"  # ✅ CORRECT - Instance variable (unique to each object)

test1 = MedicalTest("Blood Test")
test2 = MedicalTest("X-Ray")
test1.result = "Normal"

print(test2.result)  # Prints "Pending" - as expected!
```

**Rule:** Define variables unique to each object inside `__init__()` using `self`.

# Mistake 3: Calling Methods Without Parentheses

**The Trap:**

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

    def get_status(self):
        return "Healthy"

patient = Patient("John")
print(patient.get_status)  # ❌ WRONG - Prints memory address, not "Healthy"
```

**Why It's a Trap:** Without parentheses `()`, you're referencing the method object itself, not calling it to execute.

**How to Avoid:**

```
print(patient.get_status())  # ✅ CORRECT - Calls the method
```

## Mistake 4: Modifying `self` Instead of Creating Attributes

**The Trap:**

```
class Patient:
    def __init__(self, name, age):
        name = name  # ❌ WRONG - Just a local variable
        age = age    # ❌ WRONG - Disappears after __init__

patient = Patient("Alice", 30)
print(patient.name)  # AttributeError: 'Patient' has no attribute 'name'
```

**Why It's a Trap:** Without `self.`, you're creating temporary local variables that vanish when `__init__()` finishes.

**How to Avoid:**

```
class Patient:
    def __init__(self, name, age):
        self.name = name  # ✅ CORRECT - Stored in the object
        self.age = age    # ✅ CORRECT - Persists

patient = Patient("Alice", 30)
print(patient.name)  # Prints "Alice"
```

## Mistake 5: Not Understanding When `__init__` Runs {#mistake-5-not-understanding-when-init-runs }

**The Trap:**

```
class Patient:
    def __init__(self):
        print("Patient created!")

# Just defining the class - nothing prints yet
patient = Patient()  # NOW it prints "Patient created!"
```

**Why People Mess Up:** They think `__init__()` runs when you define the class. It doesn't - it only runs when you create an object.

**Concept Check:**

```
class Test:
    def __init__(self, name):
        self.name = name
        print(f"{name} initialized")

# No output yet...
test1 = Test("Blood")    # Prints "Blood initialized"
test2 = Test("Urine")    # Prints "Urine initialized"
```

Each time you instantiate, `__init__()` runs again with fresh data.

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

## How OOP Fits Into Larger Systems

An architect views OOP as a **modular organization strategy** for complex systems. Classes are modules that encapsulate related functionality, making systems:

**Modular:** Each class is a self-contained unit. You can modify the `Patient` class without breaking the `Appointment` class.

**Scalable:** Need to track 10,000 medical tests? Create 10,000 objects from one class definition. The class handles the complexity.

**Maintainable:** When bugs appear in "how patients are stored," you know exactly where to look: the `Patient` class. No hunting through 5000 lines of spaghetti code.

## Key Trade-offs in OOP Design

**Trade-off 1: Simplicity vs. Flexibility**

**Simple Approach:**

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

**Flexible Approach:**

```
class Patient:
    def __init__(self, name, age=None, blood_type=None, allergies=None):
        self.name = name
        self.age = age
        self.blood_type = blood_type
        self.allergies = allergies or []
```

The flexible version handles more scenarios but is harder to understand at a glance. Architects choose based on: "How much variability will I actually encounter?"

**Trade-off 2: Performance vs. Organization**

Creating many small objects has overhead (memory, processing time). Sometimes a simple dictionary or list is faster. But the organizational clarity of OOP often outweighs the performance cost for medium-scale applications.

**When to use OOP:** Complex applications with interrelated entities (medical record system, game engine, e-commerce platform).

**When to skip OOP:** Simple scripts, data transformation pipelines, quick automation tools.

**Trade-off 3: Abstraction vs. Transparency**

**Highly Abstracted:**

```
patient.admit()  # What does this do? Magic!
```

**Transparent:**

```
patient.status = "admitted"
patient.admission_date = datetime.now()
patient.assign_room(room_number=101)
```

Abstraction hides complexity but can make debugging harder. Transparency is explicit but verbose. Architects balance based on team expertise and project requirements.

# Core Design Principles for Robust OOP

**Principle 1: Single Responsibility**

Each class should have ONE clear purpose. A `Patient` class manages patient data. It shouldn't also handle database connections or send emails. That's what `DatabaseManager` and `EmailService` classes are for.

**Principle 2: Start Minimal, Grow As Needed**

Don't pre-engineer. Start with:

```
class MedicalTest:
    def __init__(self, name, result):
        self.name = name
        self.result = result
```

Add methods like `is_abnormal()` or `generate_pdf_report()` only when you need them.

**Principle 3: Data Hiding (Encapsulation)**

Protect internal state. In Python, convention is to prefix "private" attributes with `_`:

```
class Patient:
    def __init__(self, name):
        self.name = name
        self._medical_history = []  # "Private" - don't access directly

    def add_to_history(self, entry):
        self._medical_history.append(entry)  # Controlled access
```

This signals "interact through methods, don't manipulate the list directly."

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

## 1. Instagram: The `Post` Class

**How It's Used:** Every photo you post on Instagram is an object created from a `Post` class.

```
class Post:
    def __init__(self, user, image, caption):
        self.user = user
        self.image = image
        self.caption = caption
        self.likes = 0
        self.comments = []
        self.timestamp = datetime.now()

    def add_like(self):
        self.likes += 1

    def add_comment(self, comment):
        self.comments.append(comment)
```

**Value Created:** Instagram handles billions of posts. Each post object encapsulates all related data (image, likes, comments) and behaviors (liking, commenting). This organization prevents chaos when

scaling to billions of users.

## 2. Uber: The `Ride` Class

**How It's Used:** Each ride you take is an instance of a `Ride` class.

```
class Ride:
    def __init__(self, driver, passenger, pickup, destination):
        self.driver = driver
        self.passenger = passenger
        self.pickup = pickup
        self.destination = destination
        self.status = "requested"
        self.fare = None

    def accept_ride(self):
        self.status = "accepted"

    def complete_ride(self, fare):
        self.status = "completed"
        self.fare = fare
```

**Value Created:** Uber's system manages millions of simultaneous rides. Each ride object tracks its own state independently. When a driver accepts your ride, only YOUR ride object updates - not every ride in the system.

## 3. Gmail: The `Email` Class

**How It's Used:** Every email in your inbox is an object.

```
class Email:
    def __init__(self, sender, recipient, subject, body):
        self.sender = sender
        self.recipient = recipient
        self.subject = subject
        self.body = body
        self.is_read = False
        self.is_starred = False
        self.labels = []

    def mark_as_read(self):
        self.is_read = True

    def add_label(self, label):
        self.labels.append(label)
```

**Value Created:** Gmail handles billions of emails. Each email object knows its own state (read/unread, starred/not starred). When you mark one email as read, it doesn't affect others. OOP makes this isolation automatic.

## 4. Netflix: The `VideoPlayer` Class

**How It's Used:** When you watch a show, Netflix creates a `VideoPlayer` object for your session.

```
class VideoPlayer:
    def __init__(self, video, user):
        self.video = video
        self.user = user
        self.current_time = 0
        self.is_playing = False
        self.quality = "auto"

    def play(self):
        self.is_playing = True

    def pause(self):
        self.is_playing = False

    def seek(self, time):
        self.current_time = time
```

**Value Created:** Netflix serves millions of concurrent viewers. Each viewer's player is independent. Your pause doesn't pause mine. Each player object manages its own playback state.

## 5. Hospital Management Systems: The `Patient` Class

**How It's Used:** Real hospitals use OOP to manage patient records.

```
class Patient:
    def __init__(self, patient_id, name, dob):
        self.patient_id = patient_id
        self.name = name
        self.dob = dob
        self.medical_history = []
        self.current_medications = []
        self.allergies = []

    def add_diagnosis(self, diagnosis):
        self.medical_history.append(diagnosis)

    def prescribe_medication(self, medication):
        if medication not in self.allergies:
            self.current_medications.append(medication)
```

**Value Created:** When a doctor pulls up your file, they see an organized view of YOUR data. The `Patient` class ensures all related information (history, medications, allergies) travels together. Methods like `prescribe_medication()` enforce business rules (checking allergies before prescribing).

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

## Why CTOs Care About OOP

### Competitive Advantage: Development Speed

A CTO at a major tech company values OOP because it accelerates development. When Facebook wants to add "Story replies," developers don't start from scratch. They extend the existing `Story` class with a `add_reply()` method. **Time to market drops from months to weeks.**

### Cost Savings: Code Reusability

Without OOP, developers write similar code repeatedly. With OOP, they write a `Payment` class once and use it for subscriptions, one-time purchases, and refunds. **One class powers 10 features** instead of writing 10 separate implementations. This reduces development costs by 30-50% in large codebases.

**Competitive Advantage: Scalability**

When your startup grows from 1,000 to 1,000,000 users, OOP-designed systems handle the load. Each user is an object. Adding more objects doesn't break the system - it's designed for it. **Competitors without OOP hit scaling walls and lose users** during growth spurts.

**Risk Mitigation: Maintainability**

The average enterprise application lives 10-20 years. OOP's encapsulation means future developers can modify the `Authentication` class without understanding the entire 500,000-line codebase. **Maintenance costs drop 40%** over the application's lifetime.

# How CTOs Evaluate OOP for Their Tech Stack

**Consideration 1: Team Expertise**

"Does my team understand OOP?" If you hire Python developers, they'll expect OOP. If you hire functional programming specialists (Haskell, Elixir), forcing OOP creates friction. CTOs hire for their chosen paradigm or train existing teams.

**Consideration 2: Problem Domain**

OOP excels at modeling real-world systems with interrelated entities:

- **Good fit:** Social networks, e-commerce platforms, games, medical systems
- **Poor fit:** ETL pipelines, scientific computing, simple automation scripts

CTOs evaluate: "Does my problem map naturally to objects?"

**Consideration 3: Integration with Existing Systems**

Most companies have legacy systems. A CTO asks: "Can my OOP code talk to our 20-year-old database?" Python's OOP integrates well with most technologies, making it a safe choice.

**Consideration 4: Performance Requirements**

For 99% of applications, OOP's slight performance overhead is irrelevant. But for high-frequency trading systems or game engines, CTOs might choose lower-level approaches (C, Rust without heavy

OOP). Python with OOP handles millions of requests per day comfortably - beyond that, architectural choices (microservices, caching) matter more than OOP vs. procedural.

**Consideration 5: Long-term Vision**

CTOs thinking 5-10 years ahead choose OOP for flexibility. When business requirements change (and they always do), OOP's modularity allows surgical updates rather than complete rewrites. **This flexibility is worth millions in avoided rewrite costs.**

# 8. The Future of OOP (What's Next?)

## Trend 1: AI-Assisted OOP Development

**What's Happening:** AI tools like GitHub Copilot and ChatGPT already generate class structures from natural language descriptions. By 2030, you'll describe "I need a patient management system" and AI will generate complete class hierarchies.

**Impact:** Junior developers will produce senior-level OOP designs. The bottleneck shifts from "can we write the code?" to "do we understand the problem?"

## Trend 2: Dynamic/Adaptive Objects

**What's Happening:** Traditional OOP has fixed structures. Emerging languages (like Python's recent enhancements) allow objects to change their structure at runtime.

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

    def __getattr__(self, attr):
        # If attribute doesn't exist, create it dynamically
        return None
```

**Impact:** Systems will handle unexpected data gracefully. When a new medical test type appears, systems won't crash - they'll adapt.

## Trend 3: Distributed Objects

**What's Happening:** Cloud computing is pushing OOP to work across servers. A `User` object might have attributes stored on 5 different machines worldwide for speed.

**Impact:** You'll call `user.get_order_history()` and not know (or care) that data is coming from a server in Singapore. OOP abstracts away geographical complexity.

## Trend 4: OOP Meets Functional Programming (Hybrid Paradigms)

**What's Happening:** Modern Python encourages mixing OOP with functional concepts (list comprehensions, lambda functions, immutability).

```
class DataProcessor:
    def __init__(self, data):
        self.data = data

    def process(self):
        # Functional-style processing inside OOP
        return [transform(x) for x in self.data if validate(x)]
```

**Impact:** You'll see "best of both worlds" architectures - OOP for structure, functional for transformations.

## Trend 5: OOP in Edge Computing and IoT

**What's Happening:** Smart devices (watches, home sensors) run code locally. Each device runs lightweight OOP models.

```
class SmartHeartRateMonitor:
    def __init__(self):
        self.heart_rate = None

    def detect_anomaly(self):
        if self.heart_rate > 120:
            self.send_alert()
```

**Impact:** Your smartwatch will run its own `HealthMonitor` object, making split-second decisions without internet connectivity. OOP becomes personal and portable.

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

## How AI Helps You Learn OOP Faster

**Prompt 1: Instant Class Generation**

Use this prompt:

```
"Generate a Python class for [topic] with __init__, 3 attributes, and 2 methods. Explain each pa
```

Example: "Generate a Python class for tracking gym workouts with **init**, 3 attributes, and 2 methods."

**Result:** AI writes the code, you study the structure. Iterate until you can predict what it'll generate.

**Prompt 2: Bug Hunting**

Paste broken code and ask:

```
"Why doesn't this Python class work? Explain the error and fix it."
```

**Result:** AI becomes your debugging tutor, explaining not just the fix but WHY it failed.

**Prompt 3: Real-World Examples**

Ask:

```
"Show me how a [company] might use a Python class for [feature]. Include attributes and methods.
```

Example: "Show me how Spotify might use a Python class for playlists."

**Result:** You see practical applications, connecting theory to real products.

## Tasks You Can Automate with AI

**Automated Code Review:**

```
"Review this class for OOP best practices. Suggest improvements."
```

AI identifies poor naming, missing docstrings, or violation of single-responsibility principle.

**Generate Test Cases:**

```
"Create 5 test cases for this Patient class to verify __init__ and methods work correctly."
```

AI writes the tests, you run them to verify your code.

**Refactoring Assistance:**

```
"Refactor this procedural code into an OOP design with appropriate classes."
```

AI converts spaghetti code into organized classes, teaching you refactoring patterns.

# How AI Helps You Practice OOP

**Practice Strategy 1: Incremental Challenges**

Tell AI: "Give me a beginner Python class exercise." Complete it. Then: "Make it 10% harder." Repeat 10 times. You gradually build skill without overwhelming yourself.

**Practice Strategy 2: Code Comparison**

Write a class yourself. Ask AI: "Write a class for the same purpose." Compare. Ask: "Why is your version different from mine?" Learn from the differences.

**Practice Strategy 3: Debugging Practice**

Ask AI: "Generate a Python class with 3 subtle bugs." Find and fix them. Check with: "Verify my fixes." This builds your debugging instinct.

# How AI Helps You Design With OOP

**Design Prompt:**

```
"I'm building [project]. What classes should I create? For each class, suggest attributes and me
```

Example: "I'm building a library management system. What classes should I create?"

**Result:** AI provides architectural guidance - the hardest part of OOP. You see how experts structure systems.

# 10. Deep Thinking Triggers

### Trigger 1: The Ship of Theseus Paradox

If you create a `Car` object and replace every attribute (color, model, year) over time, is it still the same object? How does object identity work? Explore Python's `id()` function to understand object memory addresses.

### Trigger 2: The God Object Anti-Pattern

What happens when you create one massive `Application` class that does everything? Why does this violate OOP principles? How would you refactor it? This challenges you to think about boundaries and responsibilities.

### Trigger 3: Inheritance vs. Composition

If a `Car` needs an `Engine`, should `Car` inherit from `Engine` or have an engine as an attribute? When should classes inherit vs. contain other classes? Research "favor composition over inheritance."

### Trigger 4: The Immutability Challenge

Can you create a class where attributes can't be changed after initialization? Why would you want this? Explore Python's `@property` decorator and `namedtuples`.

```
class ImmutablePatient:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name
    # No setter - can't change name after creation!
```

### Trigger 5: Class vs. Dictionary

When should you use a class instead of a dictionary? Both store data. Compare:

```
# Dictionary
patient = {"name": "John", "age": 30}

# Class
class Patient:
    def __init__(self, name, age):
        self.name = name
        self.age = age
patient = Patient("John", 30)
```

What advantages does each have? When does the extra code of a class pay off?

### Trigger 6: The Factory Pattern

What if creating objects is complex? Could you create a `PatientFactory` class whose job is to create `Patient` objects? When is this useful?

```
class PatientFactory:
    @staticmethod
    def create_from_database(patient_id):
        # Fetch data from DB and create Patient object
        pass

    @staticmethod
    def create_from_form(form_data):
        # Parse form and create Patient object
        pass
```

### Trigger 7: The Philosophical Question

Are classes a natural reflection of reality, or just a mental model humans invented? Does the universe "think" in objects? How would aliens structure code?

# 11. Quick-Reference Cheatsheet

| Concept / Term | Key Takeaway / Definition |
|---|---|
| **Class** | A blueprint/template for creating objects. Defined with `class ClassName:` |
| **Object (Instance)** | A specific realization of a class with actual data. Created with `obj = ClassName()` |

| Concept / Term | Key Takeaway / Definition |
| --- | --- |
| `__init__()` | Constructor method that runs automatically when an object is created. Used to initialize attributes |
| `self` | Represents the current instance. Must be the first parameter in all instance methods |
| **Attribute** | A variable that belongs to an object. Created with `self.attribute_name = value` |
| **Instance Attribute** | Variable unique to each object. Defined inside `__init__()` with `self.variable` |
| **Class Attribute** | Variable shared by all objects of a class. Defined directly in the class body |
| **Method** | A function that belongs to a class. Always has `self` as first parameter |
| **Instance Method** | Method that operates on a specific object and can access/modify its attributes |
| **Instantiation** | The process of creating an object from a class: `my_obj = MyClass()` |
| `None` | Python's null value, representing "no value" or "nothing" |
| **Constructor Call** | Happens automatically when you create an object. You pass arguments, `__init__()` receives them |
| **Encapsulation** | Bundling data (attributes) and behavior (methods) together in one class |
| `self` **is automatic** | When calling `obj.method()`, Python automatically passes `obj` as `self` |
| **Common Error: Missing** `self` | Forgetting `self` in method definition causes `TypeError: method() takes 0 positional arguments but 1 was given` |
| **Common Error:** `self` **without dot** | Writing `name = name` instead of `self.name = name` creates a local variable that disappears |
| **Attribute Access** | Use dot notation: `obj.attribute` to read, `obj.attribute = value` to write |
| **Method Call** | Always use parentheses: `obj.method()` not `obj.method` |
| **Multiple Objects** | Each object from the same class is independent with its own attribute values |

| Concept / Term | Key Takeaway / Definition |
| --- | --- |
| **Design Principle** | One class = one clear responsibility. Don't make "god objects" that do everything |

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21