Object-Oriented Programming (OOP) in Python

System-Design Learning Notes (Very Detailed, Simple Language)

> **Goal:** Learn OOP as a way to **design reliable systems**, not as Python syntax.

---

1 WHY — Why does OOP exist? (System Purpose)

The real problem OOP solves

Before OOP, programs were written like this:

- Variables everywhere
- Functions everywhere
- Any function could modify any data
- No clear structure

This causes **system failure** when programs grow.

Problems without OOP

❌ Data corruption ❌ Hard to track bugs ❌ One change breaks many things ❌ No clear ownership ❌ Difficult to scale

System-level issue

> Large systems need **order**, **ownership**, and **boundaries**.

OOP exists to:

- Group related data + behavior
- Assign responsibility
- Protect internal state
- Reduce accidental damage

Real-life analogy (very important)

Example: Hospital system

Without OOP:

- Patient name in one place
- Age in another place
- Medical history in random files
- Any staff can change anything

Chaos ❌

With OOP:

- A **Patient object**
    - Stores patient data
    - Controls how data changes
    - Exposes only allowed actions

Order ☑

Key designer question

> "If I remove OOP, can I safely manage complex logic?"

Answer:

- Small scripts → yes
- Real systems → ❌ no

---

2 WHERE — Where does OOP live inside a system?

OOP is not everywhere. It has a **specific role**.

OOP's primary roles

System Role	What OOP does
📦 State	Stores data safely
🧠 Responsibility	Decides who owns what

System Role	What OOP does
🎛 Control	Controls how data changes
🚧 Boundary	Protects internals

OOP is the **core layer** of a system.

What OOP SHOULD touch

☑ Business logic ☑ Domain models (User, Order, Car, Patient) ☑ Rules and behavior ☑ Internal system decisions

---

What OOP should NEVER touch

❌ User input ❌ Printing / UI ❌ Network calls ❌ Database queries

> Objects should **think**, not **talk**.

Mental system diagram

```
Input → Validation → [ OOP Objects ] → Output
```

Objects sit in the **middle**, protecting logic.

---

3 WHAT CAN GO WRONG — Failure modes (Critical)

This section makes you a **real engineer**.

Common beginner mistakes

❌ Everything becomes a class

- Even small helper logic
- Leads to complexity

❌ God objects

- One class does everything
- Huge, unmaintainable

❌ Mixing responsibilities

- Input + logic + output in one class

❌ Overusing inheritance

- Deep inheritance trees
- Hard to understand behavior

❌ No encapsulation

- Data changed from anywhere

Real production failures

- One bug breaks many modules
- Impossible to test objects alone
- Tight coupling between classes
- Fear of changing code

How good systems respond

☑ Small classes ☑ Clear ownership ☑ Private internal state ☑ Explicit public methods
☑ Fail early if misused

> A designer asks: “What happens when someone uses this incorrectly?”

---

4 HOW — Minimal Python mechanism (Deep but Simple)

Now we build OOP **step by step**.

🧱 Core Idea of OOP

> **Combine data + behavior into one unit.**

That unit is an **object**.

4.1 Class — Blueprint of a responsibility

What is a class?

A class is:

- A **design**
- A **blueprint**
- A **definition of responsibility**

```
class Car:
    pass
```

Nothing is created yet.

Mental model

> "This is what a Car knows and does."

4.2 Object — A real instance

```
c1 = Car()
```

Now:

- Memory is allocated
- A real object exists

Mental model

Concept	Meaning
Class	Blueprint
Object	Real thing

4.3 `__init__` — Object construction

```

```

```
class Car:
    def __init__(self, brand):
        self.brand = brand
```

What happens step by step

1. Object is created
2. `__init__` runs automatically
3. Data is stored inside the object
4. Each object has its own copy

4.4 `self` — The object itself

`self` means:

> "This current object"

```
self.brand = brand
```

- Data belongs to **this object only**
- Not shared unless explicitly done

4.5 Attributes — Object state

Attributes:

- Represent **state**
- Store information

```
c1.brand
```

> Attributes answer: "What does this object remember?"

4.6 Methods — Object behavior

```
class Car:
    def start(self):
```

```
        print("Car started")
```

Methods:

- Act on object data
- Define allowed actions

Golden OOP Rule

> **Data should never be changed directly from outside.**

---

5 Encapsulation — Protect internal state

Why encapsulation exists

Without it:

- Anyone can modify data
- System becomes unsafe

Python implementation

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    def deposit(self, amount):
        self.__balance += amount
```

Meaning of __

- Internal detail
- Not meant for direct access

Mental model

> "You cannot touch internal wires."

---

6 Inheritance — Controlled reuse

Why inheritance exists

- Avoid duplication
- Extend existing behavior

Example

```
class Vehicle:
    def move(self):
        print("Moving")

class Car(Vehicle):
    pass
```

Car automatically gets `move()`.

Correct use rule

☑ Use inheritance only when:

> "Car **is a** Vehicle"

❌ Do NOT use for:

> "Car **has a** Engine"

Danger of inheritance

- Tight coupling
- Hidden behavior

---

7 Polymorphism — Same action, different behavior

Why polymorphism exists

Allows systems to:

- Treat different objects uniformly
- Reduce conditionals

Example

```
class Dog:
    def speak(self):
        print("Bark")

class Cat:
    def speak(self):
        print("Meow")
```

System just calls `speak()`.

System benefit

- Flexible
- Extensible
- Clean design

---

8 Abstraction — Enforcing structure

Why abstraction exists

- Force incomplete systems to fail early
- Define contracts

Example

```
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def pay(self):
        pass
```

Any subclass MUST implement `pay()`.

Mental model

> "You promised this behavior — now fulfill it."

---

9 WHO — Ownership & responsibility

Who SHOULD use OOP

☑ Core logic ☑ Domain models ☑ Business rules

Who should NOT

❌ UI ❌ Input parsing ❌ Printing ❌ Database logic

Design rule

> "If logic leaks outside objects, the system rots."

🧠 One-Line Mental Models (Very Important)

- **Class** → "A responsibility blueprint"
- **Object** → "A self-contained unit"
- **Encapsulation** → "Safety lock"
- **Inheritance** → "Controlled reuse"
- **Polymorphism** → "Same command, different result"
- **Abstraction** → "A rule without implementation"

🧠 Final System-Designer Summary

> **OOP in Python is: "A way to build systems using protected, self-contained units that own their data, control behavior, and cooperate safely."**

☑ How YOU should study this (very important)

1. Read these notes **before tutorials**
2. Watch a **simple OOP tutorial**
3. Map tutorial code → these concepts
4. Build **small systems**, not examples
5. Debug OOP mistakes deliberately
