1.1.1 Advanced Object-Oriented Programming (OOP)

(System Thinking · Real World · Production Cheatsheet)

🔹 ONE-LINE TRUTH (Never Forget)

> **OOP = Turning messy real-world systems into clean, reusable machines.**

Not theory.
Not “Java-style classes”.
It’s about **containing chaos**.

🧠 SIMPLE EXPLANATION (Mental Model)

Instead of writing **one long fragile script**, you create **Blueprints (Classes)**.

- Blueprint = **Design**
- Object = **Running Machine**
- Class = *What it should be*
- Object = *What actually exists right now*

Real-world analogy

- **Class** → Factory blueprint of a Robot
- **Object** → Actual robot working on the floor
- One blueprint → 100 robots → zero code duplication

🎯 WHY THIS MATTERS FOR *YOU*

You are building **systems**, not scripts.

You will **always** need:

- `Scraper` blueprint
- `User` blueprint
- `Question` blueprint
- `Test` blueprint
- `Pipeline` blueprint

Without OOP:

- Code becomes untestable
- One change breaks everything
- You can't scale to async, DB, APIs

👉 **OOP is how you future-proof code.**

🧩 SYSTEM THINKING FIRST (ALWAYS ASK THIS)

Before writing a class, ask:

1. **What is the real-world entity?**
   (User, Scraper, Question, Session)
2. **What state does it carry?**
   (URL, headers, data, user_id)
3. **What actions does it perform?**
   (fetch, parse, validate, save)
4. **What must NOT leak outside?**
   (passwords, tokens, internals)

If you can't answer these → don't code yet.

🔑 CORE OOP CONCEPTS (WITH THINKING + SYNTAX)

1️⃣ Class vs Object (Instance)

Mental Model

- **Class** = Recipe
- **Object** = Cooked dish

You define **behavior once**, reuse forever.

Syntax

```
class Scraper:
    pass

s = Scraper()
```

Thinking Rule

- **Classes describe**
- **Objects do**

❌ “I’ll just write functions”
✅ “This thing has identity and state → it deserves a class”

2️⃣ `__init__` — The Constructor (State Definition) {# 2️⃣ -init--the-constructor-state-definition }

What it really is

> The **birth moment** of an object.

It defines:

- What the object **knows**
- What it **starts with**

Syntax

```
class Scraper:
    def __init__(self, base_url):
        self.base_url = base_url
        self.data = []
```

Rules (Non-Negotiable)

- ❌ No business logic
- ❌ No network calls
- ❌ No parsing
- ✅ Only **state setup**

Thinking

If `__init__` breaks → your whole system breaks.

3️⃣ `self` — The Current Machine

Mental Model

> `self` = “THIS object, not others”

Without `self`, Python doesn’t know **which instance** you mean.

Syntax

```
self.data.append(item)
```

Common Mistake

```
data.append(item)  # ❌ global, dangerous
```

Thinking Rule

If it **belongs to the object**, it lives on `self`.

4️⃣ Inheritance — Parent → Child (Reuse, Not Copy)

Mental Model

> Child **is a type of** Parent

Example:

- `BaseScraper` → generic logic
- `BookScraper` → site-specific logic

Syntax

```
class BaseScraper:
    def fetch(self):
        pass

class BookScraper(BaseScraper):
    def fetch(self):
        print("Book fetch")
```

Thinking Rules

- Inherit **behavior**, not data
- Avoid deep inheritance trees
- Prefer **composition** unless reuse is clear

❌ Inheritance for “convenience”
✅ Inheritance for **shared contracts**

5️⃣ Encapsulation — Protect the System

Mental Model

> Users should not touch engine parts.

Encapsulation = **hiding internal state**.

Syntax

```
class User:
    def __init__(self, password):
        self.__password = password
```

Reality Check

- Python doesn't truly enforce privacy
- This is **intent signaling**, not security

Thinking Rule

If touching this breaks the system → hide it.

6️⃣ Polymorphism — Same Action, Different Behavior

Mental Model

> Same button, different machine reaction

Example:

- `fetch()` exists everywhere
- Behavior changes per scraper

Syntax

```
class BaseScraper:
    def fetch(self):
        raise NotImplementedError

class BookScraper(BaseScraper):
    def fetch(self):
        print("Fetching books")
```

Why this matters

- Lets you swap components
- Enables plugins
- Critical for scaling systems

❌ `if type == ...` everywhere
✅ Let objects decide their behavior

🧠 DESIGN RULES YOU SHOULD MEMORIZE

🚫 What OOP is NOT

- Not about “using classes everywhere”
- Not about long inheritance chains
- Not about clever patterns

✅ What OOP IS

- Clear ownership of data
- Predictable behavior
- Replaceable parts

- Debuggable systems

🔥 WHEN YOU ARE STUCK — ASK THIS

Symptom	Diagnosis
Too many function arguments	Missing class
Global variables everywhere	State not encapsulated
`if/else` explosion	Missing polymorphism
Hard to test	Logic + IO mixed
One change breaks all	No boundaries

🧠 FINAL MASTER MENTAL MODEL (WRITE THIS)

> **Classes are contracts.**
> **Objects are workers.**
> **Methods are actions.**
> **State lives inside.**
> **Chaos stays outside.**