## **OOP Learning Roadmap**

**We’ll go in this order:**

1. **Classes & Objects → the foundation**
2. **Attributes & Methods → properties and behaviors**
3. **Constructor (\_\_init\_\_) and self keyword**
4. **Encapsulation → data hiding and access control**
5. **Inheritance → reusing and extending functionality**
6. **Polymorphism → one interface, many forms**
7. **Abstraction → hiding complex implementation**
8. **Class vs Instance Variables & Methods**
9. **Magic / Dunder Methods (like \_\_str\_\_, \_\_len\_\_, etc.)**
10. **Real-World OOP Project (like a Banking System, Student Management, or E-commerce Cart)**

**What Are Programming Paradigms?**

A **programming paradigm** is basically a *style or approach* to writing and organizing code — a way of thinking about how to solve problems.

Think of paradigms as “different mindsets”:

* Procedural → *Step-by-step instructions (how to do it)*
* Object-Oriented → *Represent real-world things (what it is)*
* Functional → *Describe transformations (what needs to be done)*

Each paradigm solves problems in a different way, and modern languages (like Python, JavaScript, etc.) often support **multiple paradigms**.

**1. Procedural Programming**

**Concept:** It’s all about writing *procedures* (or functions) that operate on data.  
 You break down a big task into smaller steps and execute them in order.

**Think of it like:** A recipe — a set of instructions you follow in sequence.

**Example (Python):**

# Procedural approach

def calculate\_area(radius):

return 3.14 \* radius \* radius

def display\_area(radius):

area = calculate\_area(radius)

print(f"Area of circle: {area}")

radius = 5

display\_area(radius)

**Use Cases:**

* Simple scripts
* Small tools or utilities
* Data processing or automation tasks

**Advantages:**

* Easy to start with
* Great for small programs
* Direct and straightforward

**Disadvantages:**

* Hard to maintain as the program grows
* Data and logic are *separate* (can lead to tight coupling)
* Difficult to model real-world entities

**2. Object-Oriented Programming (OOP)**

**Concept:** Focuses on *objects* — which bundle **data (attributes)** and **behavior (methods)** together.

You model real-world entities like “Car,” “Customer,” or “Employee” as objects with properties and actions.

**Think of it like:** You’re building mini-worlds of interacting objects.

**Example (Python):**

# Object-Oriented approach

class Circle:

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* self.radius \* self.radius

circle1 = Circle(5)

print("Area of circle:", circle1.area())

**Use Cases:**

* Large applications (banking, e-commerce, games)
* Systems with many entities interacting (CRM, ERPs)
* When you need reusability, scalability, or abstraction

**Advantages:**

* Easier to maintain and extend (via inheritance, polymorphism)
* Models real-world entities naturally
* Code reusability through classes
* Encapsulation improves data safety

**Disadvantages:**

* More complex to design initially
* Might be overkill for small scripts

**3. Functional Programming**

**Concept:** Treats computation as *evaluations of pure functions* — no changing state, no side effects.

**Think of it like:** You give inputs, you get outputs. That’s it. No modifying variables in between.

**Example (Python):**

# Functional approach

from math import pi

area = lambda r: pi \* r \* r

print("Area of circle:", area(5))

Or using **map, filter, reduce**:

numbers = [1, 2, 3, 4]

squares = list(map(lambda x: x\*\*2, numbers))

print(squares) # [1, 4, 9, 16]

**Use Cases:**

* Data transformations (ETL, analytics pipelines)
* Parallel or distributed computing
* When predictability and immutability matter (e.g. finance, AI)

**Advantages:**

* Easier to test and debug (no side effects)
* Short, expressive code
* Great for concurrent programming

**Disadvantages:**

* Can be abstract/hard to understand for beginners
* Doesn’t model real-world objects well

## **Major Differences: OOP vs Procedural vs Functional**

| **Feature** | **Procedural** | **Object-Oriented (OOP)** | **Functional** |
| --- | --- | --- | --- |
| **Focus** | Functions & procedures | Objects (data + behavior) | Pure functions & data transformations |
| **Data Handling** | Separate from functions | Bundled with objects | Immutable |
| **Code Reuse** | Via functions | Via inheritance & polymorphism | Via composition of functions |
| **State Management** | Global or local variables | Encapsulated in objects | Avoided (no mutable state) |
| **Real-world Modeling** | Not natural | Very natural | Not focused on real-world entities |
| **Ease of Testing** | Moderate | Moderate | Easy (pure functions) |
| **Use Case** | Simple scripts | Large systems | Data transformations |

## **So Why Use OOP If Procedural Can Do the Same?**

You’re right — **you *can* achieve almost anything procedurally**.  
 But OOP is not about *capability*, it’s about *manageability*.

Let’s see a simple analogy

### **Example: Building a Car Management System**

**Procedural way:**

car\_name = "Tesla"

car\_speed = 120

def accelerate(speed):

return speed + 10

car\_speed = accelerate(car\_speed)

**If you now add 100 cars — it becomes chaos with multiple variables and functions managing each one.**

**OOP way:**

class Car:

def \_\_init\_\_(self, name, speed):

self.name = name

self.speed = speed

def accelerate(self):

self.speed += 10

car1 = Car("Tesla", 120)

car2 = Car("BMW", 100)

car1.accelerate()

print(car1.speed)

**Now each car carries its own data and methods, making it easy to manage and extend.**

**OOP shines in scalability and abstraction, not just functionality.**

## **When to Use What?**

| **Situation** | **Best Paradigm** | **Why** |
| --- | --- | --- |
| Small automation script | Procedural | Quick, simple, readable |
| Large software system (e.g., ecommerce app, game engine) | OOP | Scalable, modular, maintainable |
| Data transformation pipeline or ML preprocessing | Functional | Concise, stateless, easier to parallelize |
| API or Web framework | OOP (with some functional) | Classes to model entities, but functions to handle logic |

**Summary**

| **Paradigm** | **Mindset** | **Example** |
| --- | --- | --- |
| Procedural | "How do I do it step by step?" | Cooking recipe |
| OOP | "What entities exist and how do they interact?" | Managing cars, customers |
| Functional |  | Converting, mapping, reducing data |

**Topic 1: Classes and Objects**

Let’s start from the base.

In Python (and in OOP generally):

* A **Class** is a *blueprint* for creating objects.
* An **Object** is an *instance* of a class — a real thing built from that blueprint.

Think of:

* Class → “Car blueprint”
* Object → “Tesla Model 3”, “BMW i8”

Each object can have:

* **Attributes (variables)** → define properties (e.g., color, speed)
* **Methods (functions)** → define behavior (e.g., drive(), brake())

### **Example 1: The Simplest Class**

class Car:

pass # an empty class for now

# Creating objects

car1 = Car()

car2 = Car()

print(car1)

print(car2)

This just creates two objects from the same blueprint.  
 Even though they’re empty, Python still knows they belong to the Car class.

**Example 2: Adding Attributes & Methods**

class Car:

def \_\_init\_\_(self, brand, color):

self.brand = brand

self.color = color

def start\_engine(self):

print(f"{self.brand} is now running!")

# Creating objects

car1 = Car("Tesla", "Red")

car2 = Car("BMW", "Blue")

# Accessing attributes

print(car1.brand) # Tesla

print(car2.color) # Blue

# Calling method

car1.start\_engine() # Tesla is now running!

### **Key Points**

| **Concept** | **Description** |
| --- | --- |
| **Class** | A blueprint or template for creating objects |
| **Object** | An instance created from the class |
| **Attributes** | Data associated with the object |
| **Methods** | Functions that operate on object data |
| **\_\_init\_\_ method** | Constructor — called automatically when you create an object |
| **self** | Refers to the current instance (object) |

**Use Case: Modeling a Real-world Entity**

Let’s say you’re working in a **ride-sharing company (like Uber)**.  
 You might represent each driver as an object.

class Driver:

def \_\_init\_\_(self, name, rating):

self.name = name

self.rating = rating

def accept\_ride(self):

print(f"{self.name} has accepted the ride!")

def update\_rating(self, new\_rating):

self.rating = new\_rating

print(f"{self.name}'s new rating: {self.rating}")

# Using it

driver1 = Driver("John", 4.8)

driver2 = Driver("Emma", 4.5)

driver1.accept\_ride()

driver2.update\_rating(4.7)

Here:

* Each driver is an **object**
* They share the same **class structure**
* Each can have unique **data and actions**

**Practice Questions:**

Let’s test what you’ve learned before moving forward:

**Q1.** What’s the difference between a *class* and an *object*?  
 **Q2.** Why do we use self in class methods?  
 **Q3.** What does the \_\_init\_\_ method do?  
 **Q4.** Write a small class called Book that has attributes: title, author, and price.

* Add a method display\_info() that prints these details.
* Then, create two objects of this class and call the method.