🔟

Chapter 10

🧱 Chapter 10 – Object Oriented Programming (OOP) in Python

📘 1. What is a Class?

```
class Employee:  # 🏗️ This creates a new class blueprint
    name = "Prathamesh"  # 🏷️ Class attribute
    age = 17
    salary = 10000000

print(Employee.name, Employee.age, Employee.salary)
```

🧠 Concept:

- A **class** is a blueprint or template for creating objects.
- Variables like `name`, `age`, and `salary` inside the class (outside functions) are called **class attributes**.
- These attributes are shared among all instances unless overridden.

🧍 2. Class vs Instance Attributes

```
class Employee:
    age = 17
    salary = 10000000  # 🎯 Class Attribute

prathamesh = Employee()
prathamesh.salary = 12000000  # 🧍 Instance Attribute
print(Employee.age, Employee.salary)
```

🧠 Concept:

- **Class Attributes**: Belong to the class itself (shared across all objects).
- **Instance Attributes**: Unique to each object (created using `object.attribute = value`).
- 🔁 If both exist, **instance attribute overrides** class attribute during access.

🧠 3. The `self` Keyword – Instance Method

```
class Employee:
    age = 17
    salary = 10000000  # 🎯 Class Attribute

    def getInfo(self):  # 👀 'self' refers to current object
        print(f"The age is {self.age} and salary is {self.salary}")

    @staticmethod
    def greet():  # 🚫 Doesn't need 'self', doesn't touch object data
        print("Good Morning")

prathamesh = Employee()
prathamesh.salary = 12000000

Employee.greet()  # Static method (no object data used)
Employee.getInfo(prathamesh)  # Equivalent to prathamesh.getInfo()
```

🧠 Concept:

- `self` gives access to the object's attributes/methods inside class functions.
- Static methods don't use `self`, they behave like normal functions inside a class.

🧪 4. The `__init__()` Constructor – Dunder Method

```
class Employee:
    def __init__(self, name, age, salary):
        self.name = name  # 💡 These are instance attributes
        self.age = age
        self.salary = salary
        print("✅ Object created!")

    @staticmethod
    def greet():
        print("Good Morning")

prathamesh = Employee("Prathamesh", 18, 1300000)
prathamesh.salary = 12000000  # 🔁 Overrides instance salary

print(prathamesh.name, prathamesh.age, prathamesh.salary)
```

🧠 Concept:

- `__init__()` is the **constructor**, called **automatically** when object is created.
- Used to set initial values using **positional arguments**.
- 🎯 You *must* use `self.<name>` to assign those arguments to the object's attributes.

💡 Real Life Analogy

Concept	Analogy
Class	Blueprint of a Car 🛠️
Object	Actual Car 🚗 made using the blueprint

Attributes	Color, model, engine type of the car
Methods	Drive, start, stop
Constructor	When a new car rolls off factory with settings

🧠 Deep Dive – Advanced OOP Concepts in Python

🧬 1. Inheritance – "Reuse & Extend Code"

> 🧪 When a class inherits from another class, it gets access to all its methods and properties.

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

    def show(self):
        print(f"Employee name is {self.name}")

# 👇 Manager inherits from Employee
class Manager(Employee):
    def displayRole(self):
        print(f"{self.name} is a Manager")

m = Manager("Prathamesh")
m.show()  # Inherited
m.displayRole()  # Own method
```

🧠 Use Case:

- Reduces code duplication ✅
- Adds specialization in child class 🧠

🧬 2. Types of Inheritance

📄 Single Inheritance:

```
class A:
    def feature(self):
        print("Feature from class A")

class B(A):
    pass

b = B()
b.feature()
```

🔗 Multiple Inheritance:

```
class Father:
    def skills(self):
        print("Guitar, Cooking")

class Mother:
    def skills(self):
        print("Painting")

class Child(Father, Mother):
    pass

c = Child()
c.skills()  # ⚠️ MRO decides which method is called first
```

🧬 Multilevel Inheritance:

```
class Grandparent:
    def property(self):
        print("Land & House")

class Parent(Grandparent):
    def assets(self):
        print("Car")
```

```
class Child(Parent):
    pass

c = Child()
c.property()
c.assets()
```

🎭 3. Polymorphism – "Same Function, Different Behavior"

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

class Dog:
    def speak(self):
        print("Woof")

# Common interface
def pet_talk(pet):
    pet.speak()

pet_talk(Cat())  # Meow
pet_talk(Dog())  # Woof
```

🧠 Why it's cool:

- Makes your code flexible and reusable.
- Works well with functions that take many types.

🛡️ 4. Encapsulation – "Private Data"

> Restrict access to internal details of a class. Use getter/setter.

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # 👀 Private variable

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

    def get_balance(self):
        return self.__balance

acc = BankAccount(5000)
acc.deposit(2000)
print(acc.get_balance())  # ✅ Access via getter
# print(acc.__balance) ❌ Will throw AttributeError
```

> Use __ to make attributes private and safe from direct access.

🔁 5. Method Overriding – "Child Changes Behavior"

```
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    def greet(self):  # 🔁 Overrides A's greet
        print("Hello from B")

b = B()
b.greet()  # Calls B’s version
```

🧰 6. Special Methods (Dunder Methods)

`__str__()` – Make Objects Human-Readable

```
class Book:
    def __init__(self, title):
        self.title = title

    def __str__(self):
        return f"📖 Book: {self.title}"

b = Book("Atomic Habits")
print(b)  # 📖 Book: Atomic Habits
```

`__repr__()` – For Debugging (usually developer focused)

🧙‍♂️ 7. Class Methods – @classmethod

> Acts on the class itself not the instance.

```
class User:
    count = 0

    def __init__(self):
        User.count += 1

    @classmethod
    def get_user_count(cls):
        return cls.count

print(User.get_user_count())
```

🧼 8. Clean Object Destruction – `__del__`

```
class Person:
    def __del__(self):
        print("Object deleted... clean up here!")
```

```
p = Person()
del p  # Triggers __del__()
```

📚 Summary Table – OOP Advanced Concepts

💡 Concept	📌 Description	🧪 Keywords / Examples
Inheritance	One class derives from another	`class B(A):`
Polymorphism	Same method, different class behavior	`def speak()`
Encapsulation	Hide data using private attributes	`__balance` , `get_balance()`
Method Overriding	Redefining parent class method in child	`def greet()`
`__init__`	Constructor	Called during object creation
`__str__` / `__repr__`	Special methods for object printing/debugging	`__str__()`
Class Method	Method that works on class not instance	`@classmethod`
Static Method	Utility method not tied to object	`@staticmethod`
`__del__()`	Destructor – for object cleanup	`del object`

🚀 Real-Life Analogy

OOP Concept	Real World Analogy
Class	Blueprint of a building 🏗️
Object	Actual house built from blueprint 🏠
Inheritance	Child inherits house from parent 👴➡️👶
Encapsulation	Locking documents in a safe 🔐
Polymorphism	Button acts differently in remote vs elevator
Static Method	A calculator app that doesn't need user data

🧪 OOP Practice Problems – Python

👨‍💻 Problem 1: Microsoft Programmer Class

📁 File: `problem1.py`

```
# 🧠 Create a class programmer for storing programmers at Microsoft
class Programmer:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
        print(f"{name} is {age} years old and earns ₹{salary}")

# 👇 Creating objects
prathamesh = Programmer("Prathamesh", 17, 250000000)
harry = Programmer("Harry", 39, 20000000)
tarry = Programmer("Tarry", 23, 12000000)
```

✅ Key Concepts:

- Constructor `__init__` used to initialize object.
- Each object stores personal data: name, age, salary.
- Automatic print on creation = good for logs.

🧮 Problem 2: Basic Calculator Class

📁 File: `problem2.py`

```
# 🧠 Calculator capable of finding square, cube, square root of numbers
class Calculator:
    def __init__(self, square, cube, square_root):
        print(f"Square: {square**2}")
        print(f"Cube: {cube**3}")
        print(f"Square Root: {square_root**0.5}")

# 👇 Inputs
a = int(input("Square of: "))
b = int(input("Cube of: "))
c = int(input("Square Root of: "))
```

```
# 📦 Object
Calculator(a, b, c)
```

✅ Key Concepts:

- Basic math logic inside constructor.
- You can break this into individual methods later if needed.
- No return values → output directly.

🔁 Problem 3: Class vs Instance Attribute

📁 File: `problem3.py`

```
# 🧠 Investigate whether instance attribute changes class attribute
class Operator:
    a = 21  # 🔒 Class attribute

o = Operator()
print(o.a)  # ✅ 21 → from class
o.a = 0     # ⚠️ Creates an instance variable (doesn't modify class one)
print(o.a)  # ✅ 0 → from instance
print(Operator.a)  # ✅ 21 → original class value
```

✅ Key Concepts:

- `object.attribute = value` creates instance attribute.
- Class attribute stays unchanged unless explicitly modified like `ClassName.attribute`.

👋 Problem 4: Add Static Method to Greet

📁 File: `problem4.py`

```
# 🧠 Same calculator, now with a static method to greet
class Calculator:
    def __init__(self, square, cube, square_root):
        print(f"Square: {square**2}")
        print(f"Cube: {cube**3}")
        print(f"Square Root: {square_root**0.5}")

    @staticmethod
    def greet():
        print("👋 Hello, Welcome to Calculator!")

Calculator.greet()  # 🔔 Static methods don't need object

# 👇 Inputs
a = int(input("Square of: "))
b = int(input("Cube of: "))
c = int(input("Square Root of: "))

Calculator(a, b, c)
```

✅ Key Concepts:

- `@staticmethod` → doesn't need `self` or access to object.
- Used for utility functions related to class logic.

🚂 Problem 5: Train Booking System

📂 File: `problem5.py`

```
# 🧠 Simulate basic train booking, status, fare
from random import randint  # 🎲 Random values for dynamic behavior

class Train:
    def __init__(self, method, status, fare):
        print(f'''
Train Booking Info:
🔹 Booking Method: {method}
🔸 Available Seats: {status}
💸 Fare: ₹{fare}
''')
```

```
Train("Cash", randint(0, 100), randint(344, 7278))
```

✅ Key Concepts:

- Shows how class can be used for real-world modeling.
- `randint()` gives random values to simulate dynamic train data.

🧠 Problem 6: Can we change `self` ?

📁 File: `problem6.py`

```
# 🧠 Can you change the self parameter to something else?
class Employee:
    age = 17
    salary = 10000000

    def getInfo(sf):  # 🔁 'sf' used instead of 'self'
        print(f"Age: {sf.age}, Salary: {sf.salary}")

# 👇 Object and call
prathamesh = Employee()
prathamesh.salary = 12000000  # Creates instance attribute
Employee.getInfo(prathamesh)
```

✅ Key Concepts:

- `self` is just a naming convention; `sf` , `this` , or anything works.
- It's the *first parameter in instance methods* that refers to the object.

🧾 Chapter 10 Summary Table – Object-Oriented Programming (OOP)

🔢 Sr.	🧠 Concept / Topic	📘 Description	💡 Key Points / Examples

1	`class` Keyword	Defines a blueprint for objects	`class Employee: ...`
2	Class Attributes	Variables shared across all instances	`name = "Prathamesh"` inside class
3	Instance Attributes	Attributes specific to the instance	`self.name = name` in constructor
4	Object Creation	Instance of class using `object = ClassName()`	`e = Employee()`
5	Accessing Attributes	Using dot notation	`print(e.name)`
6	Difference: Class vs Instance Attributes	Instance attribute overrides class attribute	`obj.attr = val`
7	`self` Keyword	Refers to the current instance in methods	`def getInfo(self):`
8	`__init__()` Constructor	Special method called when object is created	`def __init__(...)`
9	Static Methods	Independent functions inside class (no `self`)	`@staticmethod`
10	Changing `self` name	`self` can be renamed (e.g., `sf` , `this`)	`def getInfo(sf):`
11	Object Initialization with Arguments	Passing values to `__init__` to initialize attributes	`Employee("Prathamesh", 18, 1300000)`
12	Practice: Programmer Class	Storing multiple programmer objects	`Programmer(name, age, salary)`
13	Practice: Calculator Class	Computes square, cube, square root	Constructor with `**` and `**0.5`
14	Practice: Class vs Instance Attribute	Instance attribute doesn't affect class attribute	`object.attr = ...`
15	Practice: Static Greet Method	Static method added to calculator	`@staticmethod def greet()`
16	Practice: Train Booking Simulation	Real-world example with dynamic fare, seat status	`randint()` used

17	Deep Dive: `__str__()`	Returns string representation of object	`def __str__(self): return ...`
18	Deep Dive: `__repr__()`	For developers/debuggers, returns more detailed string	`def __repr__(self): return ...`
19	Deep Dive: `__del__()`	Destructor method, called when object is deleted	`def __del__(self): ...`
20	Deep Dive: Inheritance	Deriving a class from another class	`class Child(Parent): ...`
21	Deep Dive: `super()`	Calls parent class methods or constructor	`super().__init__()`
22	Deep Dive: Method Overriding	Child class overrides a parent method	Redefine method in child
23	Deep Dive: Multiple Inheritance	Class derived from more than one base class	`class C(A, B): ...`
24	Deep Dive: Class Methods (`@classmethod`)	Operates on the class, takes `cls` instead of `self`	`@classmethod def set(cls):`
25	Deep Dive: Encapsulation	Bundling data and methods; access modifiers (`_` and `__`)	`_protected` , `__private`
26	Deep Dive: Polymorphism	Same method behaves differently depending on object	`len("abc")` vs `len([1,2,3])`
27	Deep Dive: Dunder Methods	Methods like `__add__` , `__len__` , etc. for operator overloading	`def __add__(self, other):`
28	Deep Dive: `isinstance()` and `issubclass()`	Type checking for objects and classes	`isinstance(obj, Class)`
29	Python Naming Conventions	`snake_case` , `CamelCase` , `self` , `cls`	Follow PEP8 where possible

✅ Extras and Observations

- **Instance vs Class** – Class attributes are shared, instance attributes are unique.
- **Static vs Class vs Instance Methods**:
  - Static → independent utility.
  - Class → modifies class-level state.
  - Instance → works on object state.
- **Good Practices**:
  - Always comment classes and methods.
  - Use `__str__()` for human-friendly printouts.
  - Avoid too much logic in `__init__()`.

🧠 Recommended Flow for Mastery

1. **Understand Syntax** (`class` , `self` , `__init__`)
2. **Create and Use Objects** (`Employee("A", 20, 5000)`)
3. **Play with Attributes** (class vs instance)
4. **Add Static / Class Methods**
5. **Practice Modeling Real-world Concepts** (Train, Programmer, Calculator)
6. **Explore Advanced Concepts** (Inheritance, Polymorphism, Encapsulation)
7. **Use Dunder Methods for Magic** (`__add__` , `__len__`)
8. **Use `isinstance()` for checks**
9. **Build Small Projects** combining these ideas