Unit 4 – Object-Oriented Programming in Python

OOP (Object-Oriented Programming) is a paradigm that organizes code into **objects** (instances) and **classes** (blueprints).
Python supports OOP fully, making it easier to model real-world entities, encourage reuse, and manage complexity.

4.1: Introduction to Classes and Objects

1. Definition / Concept

- **Class** → A **blueprint** for creating objects. Defines attributes (data) and methods (functions).
- **Object** → An **instance** of a class. Created from the blueprint, with real values.
- OOP groups **data (attributes)** and **behavior (methods)** together.

2. Analogy / Real-Life Connection

Think of **class** as a **car blueprint** in a factory:

- The blueprint describes **what every car should have** (engine, color, wheels).
- Each **car object** built from the blueprint is unique (my car = red, your car = blue).

3. Syntax

```
# Defining a class
class ClassName:
    def __init__(self, attributes):
        self.attributes = attributes

    def method(self):
        # behavior
        pass

# Creating an object
obj = ClassName(values)
obj.method()
```

4. Step-by-Step Explanation

1. **Define a class** using class ClassName:
2. **Initialize objects** using the __init__ constructor.
3. Use self to refer to the **current object**.
4. Create objects by calling the class like a function.
5. Access attributes/methods using object.attribute or object.method().

5. Example Code

(a) Simple Class & Object

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

    def drive(self):
        print(f"The {self.color} {self.brand} is driving!")

# Create objects
car1 = Car("Toyota", "Red")
car2 = Car("Tesla", "Blue")

car1.drive()
car2.drive()
```

Output:

```
The Red Toyota is driving!
The Blue Tesla is driving!
```

(b) Adding Attributes Later

```
class Student:
    pass

s1 = Student()
s1.name = "Alice"
```

```
s1.age = 20
print(s1.name, s1.age)
```

Output:

Alice 20

(c) Multiple Objects

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

    def bark(self):
        print(f"{self.name} says Woof!")

d1 = Dog("Buddy")
d2 = Dog("Charlie")

d1.bark()
d2.bark()
```

Output:

Buddy says Woof!

Charlie says Woof!

6. Diagram / Flow

📌 **Class–Object Relationship**

```
Class (Blueprint)
    |
Object 1 (instance with values)
Object 2 (instance with values)
```

📌 Example with Car:

```
Car (class) ---> car1 (Toyota, Red)
            ---> car2 (Tesla, Blue)
```

7. Output

- Already shown with each example.

- Objects can have **unique values** while sharing the same **class structure**.

8. Common Errors & Debugging

❌ **Error 1: Forgetting self in method**

```
class Test:
    def hello():
        print("Hi")

t = Test()
t.hello()  # TypeError
```

✅ Fix: Always include self as the first parameter in instance methods.

❌ **Error 2: Misusing class vs object**

```
class Test:
    value = 10

print(Test.value)    # works (class variable)
print(Test().value)  # works (object too)
```

But mixing them without understanding leads to confusion.

❌ **Error 3: Wrong initialization**

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

s = Student()   # ❌ Missing argument
```

✅ Fix: Pass required arguments → Student("Alice").

9. Interview / Industry Insight

- Interviewers often ask:
  - Difference between **class and object**.
  - Purpose of self.

- Output of code involving multiple objects.
- In industry:
  - OOP is used everywhere (bank accounts, products, customers, vehicles).
  - Large frameworks (Django, Flask) are written with OOP principles.

4.2: The __init__ Constructor

1. Definition / Concept

- The **__init__ method** in Python is a **special method (constructor)** that automatically runs **when a new object is created** from a class.
- It is used to **initialize attributes** (set initial values).
- Defined using def __init__(self, ...): inside a class.

📌 Key point: __init__ is **not mandatory**, but highly recommended for setting up objects.

2. Analogy / Real-Life Connection

Think of **__init__** as the **welcome ceremony** for a newborn baby:

- When the baby (object) is born, they are automatically given a **name, DOB, weight**.
- Similarly, when an object is created, __init__ automatically assigns **initial values**.

3. Syntax

```
class ClassName:
    def __init__(self, param1, param2, ...):
        self.param1 = param1
        self.param2 = param2
```

- __init__ → special method name.
- self → refers to the **current object**.
- Parameters passed to the class are used to **initialize attributes**.

4. Step-by-Step Explanation

1. When you call obj = ClassName(args), Python internally:
   - Creates a new object.
   - Calls __init__ automatically with the provided arguments.
2. Inside __init__, use self.attribute = value to initialize.

3. self ensures that each object keeps **its own separate data**.

5. Example Code

(a) Basic Constructor

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

p1 = Person("Alice", 25)
print(p1.name, p1.age)
```

Output:

```
Alice 25
```

(b) Constructor with Default Values

```
class Student:
    def __init__(self, name, course="Python"):
        self.name = name
        self.course = course

s1 = Student("John")
s2 = Student("Jane", "Java")

print(s1.name, s1.course)
print(s2.name, s2.course)
```

Output:

```
John Python
Jane Java
```

(c) Using Methods after Initialization

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
```

```
        return 3.14 * self.radius ** 2

c1 = Circle(5)
print("Area:", c1.area())
```

Output:

Area: 78.5

(d) Multiple Objects with Different Data

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

car1 = Car("Tesla", "Red")
car2 = Car("BMW", "Black")

print(car1.brand, car1.color)
print(car2.brand, car2.color)
```

Output:

Tesla Red

BMW Black

6. Diagram / Flow

📌 **Constructor Flow**

```
Class Definition
      |
Object Creation ---> __init__() called automatically
      |
Attributes initialized (stored inside self)
```

📌 Example with Person Class

```
p1 = Person("Alice", 25)
          |
__init__(self, "Alice", 25)
self.name = "Alice"
```

```
self.age = 25
```

7. Output

- Already shown in examples.
- __init__ **always runs automatically** when an object is created.

8. Common Errors & Debugging

❌ **Error 1: Missing arguments**

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


p = Person()   # ❌ missing argument
```

Error: TypeError: __init__() missing 1 required positional argument

✅ Fix: Provide required args → Person("Alice").

❌ **Error 2: Forgetting self**

```
class Person:
    def __init__(name):  # ❌ missing self
        self.name = name
```

Error: NameError: name 'self' is not defined

✅ Fix: Always put self first → def __init__(self, name):

❌ **Error 3: Mixing class and object attributes**

```
class Test:
    def __init__(self, value):
        value = value   # ❌ wrong, not bound to object
```

✅ Fix: Use self.value = value

9. Interview / Industry Insight

- Interviewers love asking:
  - What is __init__?
  - Difference between __init__ and normal methods.
  - Can a class have multiple __init__ methods? (Answer: No, but you can use **default args** or *args).
- In industry:
  - __init__ is used in almost every class (models in Django, Flask, data objects in ML, etc.).
  - Good constructors improve readability and reduce bugs.

4.3: Instance vs Class Variables

1. Definition / Concept

- **Instance Variables**
  - Belong to a **specific object (instance)**.
  - Defined inside the __init__ method using self.
  - Each object has its **own copy**.
- **Class Variables**
  - Belong to the **class itself**.
  - Shared across **all objects**.
  - Defined directly inside the class (but outside methods).

📌 Key: Instance variables differ per object, class variables are shared.

2. Analogy / Real-Life Connection

- **Class variable** → Like a **school name** → all students share the same school.
- **Instance variable** → Like a **student's name/age** → unique to each student.

3. Syntax

```
class ClassName:
    class_var = "I am a class variable"

    def __init__(self, value):
        self.instance_var = value  # instance variable
```

4. Step-by-Step Explanation

1. **Class Variables** are defined at the class level → accessible via both class and objects.
2. **Instance Variables** are created inside __init__ → unique to each object.
3. Changing a **class variable** affects all objects.
4. Changing an **instance variable** affects only that object.

5. Example Code

(a) Instance vs Class Variable

```
class Student:
    school = "ABC High School"  # class variable

    def __init__(self, name, age):
        self.name = name      # instance variable
        self.age = age

s1 = Student("Alice", 20)
s2 = Student("Bob", 22)

print(s1.name, s1.age, s1.school)
print(s2.name, s2.age, s2.school)
```

Output:

```
Alice 20 ABC High School
Bob 22 ABC High School
```

(b) Modifying Instance Variable

```
s1.age = 21
print("Alice age:", s1.age)
print("Bob age:", s2.age)   # unaffected
```

Output:

```
Alice age: 21
Bob age: 22
```

(c) Modifying Class Variable

```
Student.school = "XYZ International School"
```

print("Alice's school:", s1.school)

print("Bob's school:", s2.school)

Output:

Alice's school: XYZ International School

Bob's school: XYZ International School

(d) Shadowing Class Variable with Instance Variable

s1.school = "Private School" # creates instance variable 'school'

print("s1 school:", s1.school)

print("s2 school:", s2.school)

print("Class school:", Student.school)

Output:

s1 school: Private School

s2 school: XYZ International School

Class school: XYZ International School

6. Diagram / Flow

📌 **Memory View**

```
Class: Student
  ├── school = "XYZ International School"    (class variable, shared)
  │
  ├── s1 (object)
  │      ├── name = "Alice"
  │      ├── age = 21
  │      └── school = "Private School" (shadowed)
  │
  └── s2 (object)
         ├── name = "Bob"
         ├── age = 22
         └── (uses class variable school)
```

7. Output

- Already shown in examples.
- Key: Instance vars → unique, Class vars → shared unless shadowed.

8. Common Errors & Debugging

❌ **Error 1: Confusing instance and class variables**

```
class Test:
    x = 5  # class var

t1 = Test()
t1.x = 10   # creates instance var, does not modify class var
print(Test.x, t1.x)
```

Output:

```
5 10
```

✅ Fix: Use ClassName.var to modify class variables.

❌ **Error 2: Overusing class variables**

- If you mistakenly store object-specific data in class variables, all objects will share it.

✅ Fix: Keep per-object data inside __init__.

9. Interview / Industry Insight

- Interviewers often ask:
  - Difference between instance and class variables.
  - Predict output when modifying class variable vs instance variable.
- Industry:
  - **Class variables** often used for constants, configuration, counters.
  - **Instance variables** store unique object data.
  - Example: Django models → class variables define table fields, instance variables store row data.

4.4: Instance Methods, Class Methods, Static Methods

1. Definition / Concept

- **Instance Methods**

- Operate on **individual objects (instances)**.
- First parameter is always self.
- Can access/modify **instance variables** and **class variables**.

- **Class Methods**
  - Belong to the **class** rather than any object.
  - First parameter is cls (class itself).
  - Declared using @classmethod.
  - Can access/modify **class variables**, but not instance variables directly.
- **Static Methods**
  - Independent of class/object.
  - Declared using @staticmethod.
  - Behaves like a normal function inside class, but logically grouped with the class.
  - Cannot access/modify self or cls.

2. Analogy / Real-Life Connection

- **Instance Method** → Like a **student's personal diary** → specific to each student.
- **Class Method** → Like the **school announcement board** → common for all students.
- **Static Method** → Like a **general utility (calculator, weather info)** → useful, but doesn't depend on any student or school.

3. Syntax

```
class ClassName:
    # Instance method
    def instance_method(self, arg):
        ...

    # Class method
    @classmethod
    def class_method(cls, arg):
        ...

    # Static method
    @staticmethod
    def static_method(arg):
        ...
```

4. Step-by-Step Explanation

1. **Instance Methods** → Default methods, require an object to call.
2. **Class Methods** → Use @classmethod, take cls, can modify class-level data.
3. **Static Methods** → Use @staticmethod, no self or cls, used for **utility functions**.

⚠️ **Best Practice**

- Use **instance methods** when behavior depends on the object.
- Use **class methods** when behavior affects the class (like tracking count of objects).
- Use **static methods** for helper functions.

5. Example Code

(a) Instance Method

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

    def display(self):   # instance method
        print(f"{self.name} scored {self.marks} marks.")

s1 = Student("Alice", 85)
s1.display()
```

Output:

```
Alice scored 85 marks.
```

(b) Class Method

```
class Student:
    school = "ABC School"

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

    @classmethod
    def change_school(cls, new_name):
```

```
        cls.school = new_name

s1 = Student("Bob")
print("Before:", Student.school)
Student.change_school("XYZ School")
print("After:", Student.school)
```

Output:

```
Before: ABC School
After: XYZ School
```

(c) Static Method

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

print("Sum:", MathUtils.add(5, 7))
```

Output:

```
Sum: 12
```

(d) Combining All

```
class Demo:
    class_var = 0

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

    def show(self):  # instance
        print("Instance x:", self.x)

    @classmethod
    def update_class_var(cls, val):  # class
        cls.class_var = val
        print("Class variable updated:", cls.class_var)
```

```
    @staticmethod
    def greet():  # static
        print("Hello from static method!")

obj = Demo(10)
obj.show()
Demo.update_class_var(100)
Demo.greet()
```

Output:

```
Instance x: 10
Class variable updated: 100
Hello from static method!
```

6. Diagram / Flow

📌 **Method Types**

```
Instance Method   → Needs object → self → works on instance data
Class Method      → Needs class  → cls  → works on class data
Static Method     → No self/cls  → works like normal function
```

📌 **Call Example**

```
obj.instance_method()    → bound to object
Class.class_method()     → bound to class
Class.static_method()    → independent
```

7. Output

- Already shown with examples.
- Important: Class & Static methods can be called **both by object and class**, but usually called via class.

8. Common Errors & Debugging

❌ **Error 1: Forgetting decorators**

```
class Test:
    def cm(cls):   # forgot @classmethod
```

...

Called with Test.cm() → Error.

✅ Fix: Add @classmethod.

❌ **Error 2: Using self in static method**

```
class Test:
    @staticmethod
    def hello(self):  # ❌ wrong
        print(self)
```

Will not work as expected.

✅ Fix: Remove self.

❌ **Error 3: Misunderstanding scope**

- Static methods **cannot access instance/class variables directly**.

9. Interview / Industry Insight

- Common interview Qs:
  - Difference between instance, class, and static methods.
  - Output prediction when calling methods with/without decorators.
- Industry:
  - **Instance methods** → object behavior.
  - **Class methods** → factories (alternative constructors). Example: datetime.fromtimestamp().
  - **Static methods** → utility helpers, e.g., validation, math.

4.5: Inheritance (Single, Multiple, Multilevel)

1. Definition / Concept

- **Inheritance** → An OOP feature where a class (child/derived) can **reuse properties & methods** from another class (parent/base).
- Promotes **code reuse** and logical hierarchy.

📌 **Types of Inheritance in Python**:

1. **Single Inheritance** → One parent → One child.
2. **Multiple Inheritance** → Child inherits from **multiple parents**.
3. **Multilevel Inheritance** → A child class becomes parent for another child.

2. Analogy / Real-Life Connection

- **Single** → Child inherits traits (eye color, height) from **one parent**.
- **Multiple** → Child inherits features from **both parents** (like dad's height, mom's smile).
- **Multilevel** → Grandparent → Parent → Child (traits pass through generations).

3. Syntax

```
# Single inheritance
class Parent:
    ...
class Child(Parent):
    ...

# Multiple inheritance
class Parent1:
    ...
class Parent2:
    ...
class Child(Parent1, Parent2):
    ...

# Multilevel inheritance
class GrandParent:
    ...
class Parent(GrandParent):
    ...
class Child(Parent):
    ...
```

4. Step-by-Step Explanation

1. **Single Inheritance**
    - Child extends parent, inherits attributes & methods.

2. **Multiple Inheritance**
    - Child can access attributes/methods of multiple parents.
    - Python resolves conflicts using **MRO (Method Resolution Order)**.
3. **Multilevel Inheritance**
    - Chain of inheritance: Grandparent → Parent → Child.
    - Child indirectly inherits from grandparent too.

5. Example Code

(a) Single Inheritance

```
class Animal:
    def sound(self):
        print("Animals make sounds")

class Dog(Animal):
    def bark(self):
        print("Dog barks Woof!")

d = Dog()
d.sound()
d.bark()
```

Output:

```
Animals make sounds
Dog barks Woof!
```

(b) Multiple Inheritance

```
class Father:
    def skill(self):
        print("Father: Driving")

class Mother:
    def talent(self):
        print("Mother: Cooking")

class Child(Father, Mother):
    def hobby(self):
```

```
        print("Child: Painting")

c = Child()
c.skill()
c.talent()
c.hobby()
```

Output:

```
Father: Driving
Mother: Cooking
Child: Painting
```

(c) Multilevel Inheritance

```
class Grandparent:
    def legacy(self):
        print("Grandparent: Land ownership")

class Parent(Grandparent):
    def house(self):
        print("Parent: Owns a house")

class Child(Parent):
    def career(self):
        print("Child: Software Engineer")

c = Child()
c.legacy()
c.house()
c.career()
```

Output:

```
Grandparent: Land ownership
Parent: Owns a house
Child: Software Engineer
```

6. Diagram / Flow

📌 **Single Inheritance**

```
Parent → Child
```

📌 **Multiple Inheritance**

```
Parent1      Parent2
     \        /
      \      /
       Child
```

📌 **Multilevel Inheritance**

Grandparent → Parent → Child

7. Output

- Shown in examples.
- Key: Python uses **MRO (C3 linearization)** in multiple inheritance.

8. Common Errors & Debugging

❌ **Error 1: Calling parent method not present**

```
class Child:
   pass


c = Child()
c.sound()   # ❌ Error
```

✅ Fix: Ensure Child inherits → class Child(Parent):.

❌ **Error 2: Confusion in multiple inheritance**

```
class A:
   def show(self): print("A")

class B:
   def show(self): print("B")

class C(A, B): pass
c = C()
c.show()
```

Output:

A

⚠️ Because Python resolves left-to-right (A before B).

❌ **Error 3: Misunderstanding super() in multilevel**

- Using super() ensures **MRO order** is respected.

9. Interview / Industry Insight

- Common Interview Qs:
  - Types of inheritance in Python.
  - How does Python resolve conflicts in multiple inheritance? (Answer: **MRO**).
  - Difference between single vs multilevel vs multiple inheritance.
- Industry Use:
  - **Single**: Most common (e.g., Dog inherits from Animal).
  - **Multilevel**: Useful in hierarchical designs.
  - **Multiple**: Used carefully (mixins in Django, Flask).

4.6: Method Overriding and super()

1. Definition / Concept

- **Method Overriding**
  - When a **child class defines a method with the same name as the parent class method**, the child's method overrides the parent's method.
  - Provides **specific implementation** for the child class.
- **super()**
  - A built-in function that gives access to **parent class methods/attributes** from the child class.
  - Often used to call the parent's constructor (__init__) or overridden methods.

2. Analogy / Real-Life Connection

- **Method Overriding** → Like a family recipe: your **grandmother** makes curry in one way, but your **mother** makes it differently. Both are curry, but the **new version overrides** the old one.

- **super()** → Like calling your mom and saying: *"Please cook it the way grandma used to."* → You reuse parent's behavior while adding your own twist.

3. Syntax

```
class Parent:
    def show(self):
        print("Parent method")

class Child(Parent):
    def show(self):   # overriding
        print("Child method")
```

Using super():

```
class Child(Parent):
    def show(self):
        super().show()   # call parent method
        print("Child method")
```

4. Step-by-Step Explanation

1. When a child class defines a method with the **same name**, Python first checks in the child class.
2. If found, it **overrides** the parent version.
3. To still access the parent version, use super().
4. Commonly used in constructors (__init__) when parent needs initialization too.

5. Example Code

(a) Method Overriding

```
class Animal:
    def sound(self):
        print("Animals make sounds")

class Dog(Animal):
    def sound(self):   # overriding
        print("Dog barks Woof!")
```

```
d = Dog()
d.sound()
```

Output:

```
Dog barks Woof!
```

(b) Using super() to call parent method

```
class Animal:
    def sound(self):
        print("Animals make sounds")

class Dog(Animal):
    def sound(self):
        super().sound()   # call parent method
        print("Dog barks Woof!")

d = Dog()
d.sound()
```

Output:

```
Animals make sounds
Dog barks Woof!
```

(c) super() in Constructor (__init__)

```
class Person:
    def __init__(self, name):
        self.name = name
        print("Person initialized")

class Student(Person):
    def __init__(self, name, roll_no):
        super().__init__(name)   # call parent __init__
        self.roll_no = roll_no
        print("Student initialized")

s = Student("Alice", 101)
```

Output:

Person initialized
Student initialized

(d) Multiple Inheritance with super() (MRO)

```
class A:
    def show(self):
        print("A")

class B(A):
    def show(self):
        super().show()
        print("B")

class C(B):
    def show(self):
        super().show()
        print("C")

c = C()
c.show()
```

Output (follows MRO order):

A
B
C

6. Diagram / Flow

📌 **Overriding Flow**

Child.method() → if exists, override Parent.method()

📌 **super() Flow**

Child.method()
  |
  v
super().method() → Parent.method()

📌 **MRO (Method Resolution Order)**

- In multiple inheritance, Python resolves super() using **C3 Linearization** (left-to-right).
- Use ClassName.__mro__ to see order.

7. Output

- Already shown in examples.
- Key: super() always respects MRO order.

8. Common Errors & Debugging

❌ **Error 1: Forgetting super() in constructors**

```
class Child(Parent):
    def __init__(self):
        print("Child init")   # Parent init skipped
```

✅ Fix: Use super().__init__().

❌ **Error 2: Wrong assumption about multiple inheritance order**

```
class A: ...
class B: ...
class C(A, B): ...
```

Order is C → A → B → object (not alphabetical).

✅ Fix: Check using print(C.__mro__).

❌ **Error 3: Overriding without self**

```
class Test:
    def show():   # ❌ missing self
        print("Hello")
```

✅ Fix: def show(self): ...

9. Interview / Industry Insight

- Interview Qs:
  - What is method overriding?

- Difference between method overloading vs overriding (Python doesn't support overloading).
- Why use super()?
- How is super() resolved in multiple inheritance? (Answer: **MRO**).

- Industry:
  - super() is used in frameworks (Django models, Flask views).
  - Overriding is used when **customizing parent behavior** (e.g., overriding save() in Django model).

4.7: Encapsulation (Public, Protected, Private Conventions)

1. Definition / Concept

- **Encapsulation** → The OOP principle of **bundling data (attributes) and methods together** and restricting direct access to some attributes.
- In Python, encapsulation is implemented using **naming conventions** (since Python doesn't enforce strict access control like Java/C++).

📌 **Types of access modifiers in Python (by convention):**

1. **Public** → Accessible everywhere (default).
2. **Protected (_var)** → A convention that suggests "for internal use only." Can still be accessed, but not recommended.
3. **Private (__var)** → Name mangling used. Harder to access from outside the class.

2. Analogy / Real-Life Connection

- **Public** → Like the **front gate** of your house → anyone can use it.
- **Protected** → Like the **back door** → only family/friends are expected to use it.
- **Private** → Like your **locker with a password** → only you should have access.

3. Syntax

```
class MyClass:
    def __init__(self):
        self.public = "I am public"
        self._protected = "I am protected"
        self.__private = "I am private"
```

4. Step-by-Step Explanation

1. **Public Members** → Default. Accessible inside and outside class.
2. **Protected Members** → Prefixed with _. Not enforced, but signals "internal use only".
3. **Private Members** → Prefixed with __. Python does **name mangling** (_ClassName__var) to discourage external access.

⚠️ Python relies on **developer discipline**, not strict enforcement.

5. Example Code

(a) Public Members

```
class Person:
    def __init__(self, name):
        self.name = name   # public

p = Person("Alice")
print(p.name)  # accessible
```

Output:

```
Alice
```

(b) Protected Members

```
class Person:
    def __init__(self, name):
        self._name = name  # protected by convention

p = Person("Bob")
print(p._name)  # works, but not recommended
```

Output:

```
Bob
```

(c) Private Members with Name Mangling

```
class Person:
    def __init__(self, name):
        self.__name = name  # private

p = Person("Charlie")
```

print(p.__name) # ❌ AttributeError

# Access via name mangling
print(p._Person__name)

Output:

Charlie

(d) Encapsulation in Practice (Getter & Setter)

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

    def get_balance(self):
        return self.__balance   # getter

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount   # setter
        else:
            print("Invalid deposit")

a = Account(1000)
print("Balance:", a.get_balance())
a.deposit(500)
print("Updated Balance:", a.get_balance())
```

Output:

Balance: 1000

Updated Balance: 1500

6. Diagram / Flow

📌 **Encapsulation Layers**

```
Public     → Accessible everywhere
```

Protected → Accessible inside class & subclasses (not enforced)

Private   → Accessible only inside class (via name mangling outside)

📌 **Example with Account**

```
Account
 ├── __balance (private)
 ├── get_balance() → controlled access
 └── deposit()     → controlled modification
```

7. Output

- Shown with examples.
- Private members are hidden by name mangling, but can still be accessed if forced.

8. Common Errors & Debugging

❌ **Error 1: Direct access to private variable**

```
p = Person("Charlie")
print(p.__name)  # ❌ AttributeError
```

✅ Fix: Use p._Person__name (not recommended) or proper getter.

❌ **Error 2: Misunderstanding protected**

```
obj._var  # works, but against convention
```

⚠️ Protected is only a **convention**.

❌ **Error 3: Forgetting getters/setters**

- Direct access breaks encapsulation.
- Always provide controlled access for sensitive data.

9. Interview / Industry Insight

- Interview Qs:
  - What are access modifiers in Python?
  - How is private different from protected?
  - Does Python enforce encapsulation strictly? (Answer: No, relies on convention).
- Industry:
  - **Encapsulation** ensures **data security** and **controlled access**.
  - Example: Bank account → balance should not be directly modified.

- Frameworks (like Django ORM) use private/protected variables internally.

4.8: Special (Magic/Dunder) Methods

1. Definition / Concept

- **Special Methods** (also called **magic methods** or **dunder methods**) are **predefined methods in Python** that start and end with **double underscores** (__).
- They allow us to **customize the behavior of objects**.
- Example:
  - __init__ → constructor.
  - __str__ → string representation.
  - __len__ → length of object.
  - __add__ → overloading +.

📌 They provide **operator overloading** and make classes behave like built-in types.

2. Analogy / Real-Life Connection

Think of dunder methods like **secret backstage controls** in a theatre:

- Normally, you just enjoy the show (methods like print(), len(), +).
- But behind the scenes, **dunder methods control what happens** when those operators are used on objects.

3. Syntax

```
class ClassName:
  def __methodname__(self, other):
    # custom implementation
    pass
```

4. Step-by-Step Explanation

- **__init__** → Runs at object creation.
- **__str__** → Defines what print(obj) shows.
- **__len__** → Defines result of len(obj).
- **__add__** → Defines behavior of obj1 + obj2.
- **__eq__** → Defines equality (==).

- Many more (__getitem__, __iter__, __call__).

5. Example Code

(a) __str__ → String Representation

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

    def __str__(self):
        return f"{self.title} by {self.author}"

b = Book("1984", "George Orwell")
print(b)
```

Output:

```
1984 by George Orwell
```

(b) __len__ → Length of Object

```
class Course:
    def __init__(self, students):
        self.students = students

    def __len__(self):
        return len(self.students)

c = Course(["Alice", "Bob", "Charlie"])
print(len(c))
```

Output:

```
3
```

(c) __add__ → Operator Overloading

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
print(v1 + v2)
```

Output:

(6, 8)

(d) __eq__ → Equality Check

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

    def __eq__(self, other):
        return self.roll == other.roll

s1 = Student("Alice", 101)
s2 = Student("Bob", 101)
print(s1 == s2)
```

Output:

True

(e) __getitem__ → Indexing Support

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        return self.items[index]
```

ml = MyList([10, 20, 30])

print(ml[1])

Output:

20

6. Diagram / Flow

📌 **Operator → Dunder Method Mapping**

```
obj1 + obj2    →  obj1.__add__(obj2)
obj1 == obj2   →  obj1.__eq__(obj2)
len(obj)       →  obj.__len__()
str(obj)       →  obj.__str__()
obj[i]         →  obj.__getitem__(i)
```

7. Output

- Already shown in examples.
- Custom classes now behave like built-in types.

8. Common Errors & Debugging

❌ **Error 1: Forgetting to return string in __str__**

```
def __str__(self):
    print("Book object")   # ❌ Wrong
```

✅ Fix: Must return string.

❌ **Error 2: __add__ without returning object**

```
def __add__(self, other):
    return (self.x + other.x, self.y + other.y)  # returns tuple
```

✅ Fix: Return Vector object for consistency.

❌ **Error 3: Using equality without defining __eq__**

- By default, == compares memory addresses, not content.

✅ Fix: Implement __eq__.

9. Interview / Industry Insight

- Interview Qs:
  - What are dunder methods?
  - How does Python internally handle +, len(), print()?
  - Difference between __str__ and __repr__.
- Industry:
  - Dunder methods make classes **user-friendly** and integrate with Python's built-in features.
  - Widely used in data models (Django ORM models, Pandas objects).
  - Example: len(df) in Pandas → internally uses __len__.

4.9: Polymorphism and Duck Typing

1. Definition / Concept

- **Polymorphism** → "Many forms." The ability of different classes to provide **different implementations for the same method name**.
  - Example: len("abc") → returns 3 (string length).
  - len([1,2,3]) → returns 3 (list length).
- **Duck Typing** (Python-specific) → If an object **behaves like a type**, Python doesn't care about its actual class.
  - "If it walks like a duck and quacks like a duck, it's a duck."

2. Analogy / Real-Life Connection

- **Polymorphism** → "Play" button:
  - On a music player → plays songs.
  - On a video player → plays movies.
  - Same button, different behaviors.
- **Duck Typing** → In cricket, any person who **can hold a bat and hit the ball** is a "batsman," no matter if they're actually a professional or a kid.

3. Syntax

Polymorphism Example

```
for obj in [Dog(), Cat()]:
    obj.sound()   # same method, different behavior
```

Duck Typing Example

```
def make_it_quack(obj):
    obj.quack()   # works if obj has quack(), no matter class
```

4. Step-by-Step Explanation

1. **Polymorphism**:
   - Allows using the same method name for different classes.
   - Achieved via method overriding or operator overloading.
2. **Duck Typing**:
   - Python is dynamically typed.
   - No need to check object type explicitly.
   - If an object implements the expected method, it works.

5. Example Code

(a) Polymorphism with Classes

```
class Dog:
    def sound(self):
        print("Woof!")

class Cat:
    def sound(self):
        print("Meow!")

for animal in [Dog(), Cat()]:
    animal.sound()
```

Output:

```
Woof!
Meow!
```

(b) Built-in Polymorphism (len())

```
print(len("Hello"))    # string
print(len([1, 2, 3]))  # list
print(len({"a": 1, "b": 2}))  # dict
```

Output:

5

3

2

(c) Duck Typing Example

```
class Duck:
    def quack(self):
        print("Quack, quack!")

class Person:
    def quack(self):
        print("I can also quack like a duck!")

def make_it_quack(obj):
    obj.quack()

make_it_quack(Duck())
make_it_quack(Person())
```

Output:

Quack, quack!

I can also quack like a duck!

(d) Polymorphism with Operator Overloading

```
print(10 + 5)       # addition for numbers
print("Hello " + "World")  # concatenation for strings
```

Output:

15

Hello World

6. Diagram / Flow

📌 **Polymorphism Flow**

```
Animal (parent)
  ├── Dog.sound() → Woof!
```

```
  └── Cat.sound() → Meow!
```

📌 **Duck Typing Flow**

```
make_it_quack(obj)
    |
If obj has quack() → works
Else → AttributeError
```

7. Output

- Shown in examples.
- Demonstrates same function/operator working differently based on object type.

8. Common Errors & Debugging

❌ **Error 1: Object missing expected method**

make_it_quack(5) # ❌ AttributeError: 'int' object has no attribute 'quack'

✅ Fix: Ensure passed object implements required method.

❌ **Error 2: Overusing isinstance checks**

if isinstance(obj, Duck): # ❌ not Pythonic

  obj.quack()

✅ Fix: Use duck typing (just call obj.quack()).

❌ **Error 3: Forgetting polymorphism in inheritance**

- If child class doesn't override parent method, parent's version is used.

9. Interview / Industry Insight

- Interview Qs:
  - What is polymorphism?
  - Give an example of duck typing in Python.
  - How is polymorphism implemented in Python?
- Industry:
  - **Polymorphism** used in frameworks (e.g., Django models overriding save()).

- **Duck typing** makes Python flexible in design (no need for explicit interfaces).
- Example: Any object with __iter__() can be used in a for loop, regardless of class.

4.10: Abstract Base Classes (abc module`)

1. Definition / Concept

- **Abstract Class** → A class that cannot be instantiated directly. It serves as a **blueprint** for other classes.
- Contains **abstract methods** (declared but not implemented).
- Subclasses must provide their own implementation of these abstract methods.
- In Python, abstract classes are defined using the **abc module** (Abstract Base Classes).

📌 Keywords:

- ABC → Base class for abstract classes.
- @abstractmethod → Decorator to mark a method as abstract.

2. Analogy / Real-Life Connection

Think of an **abstract class** like an **architect's building plan**:

- You cannot **live inside a blueprint** (abstract class cannot be instantiated).
- But you can **construct real buildings** (concrete subclasses) based on it.

3. Syntax

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    @abstractmethod
    def method(self):
        pass
```

4. Step-by-Step Explanation

1. Import ABC and abstractmethod from abc module.
2. Define a class that inherits from ABC.
3. Use @abstractmethod to declare abstract methods.
4. Subclasses must implement all abstract methods, else they cannot be instantiated.

5. Example Code

(a) Basic Abstract Class

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Woof!"

class Cat(Animal):
    def sound(self):
        return "Meow!"

# a = Animal()  # ❌ Error: Can't instantiate abstract class
d = Dog()
c = Cat()
print(d.sound())
print(c.sound())
```

Output:

```
Woof!
Meow!
```

(b) Abstract Class with Constructor

```
from abc import ABC, abstractmethod

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

```
    @abstractmethod
    def drive(self):
        pass

class Car(Vehicle):
    def drive(self):
        return f"{self.brand} car is driving!"

v = Car("Tesla")
print(v.drive())
```

Output:

Tesla car is driving!

(c) Multiple Abstract Methods

```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)

rect = Rectangle(5, 10)
```

print("Area:", rect.area())

print("Perimeter:", rect.perimeter())

Output:

Area: 50

Perimeter: 30

6. Diagram / Flow

📌 **Abstract Class Hierarchy**

```
Abstract Class (Animal - blueprint)
      |
      ├── Dog (implements sound)
      └── Cat (implements sound)
```

📌 **Object Flow**

Animal (abstract) ❌ cannot instantiate

Dog() ✔

Cat() ✔

7. Output

- Attempting to create an object of an abstract class results in:

TypeError: Can't instantiate abstract class Animal with abstract method sound

8. Common Errors & Debugging

❌ **Error 1: Not implementing abstract method in subclass**

class Bird(Animal): # Missing sound() implementation

pass

✅ Fix: Implement all abstract methods.

❌ **Error 2: Forgetting to inherit from ABC**

class Shape:

@abstractmethod

def area(self): # ❌ Error

```
    pass
```

✅ Fix: Must inherit from ABC.

❌ **Error 3: Mixing abstract and concrete classes**

- Concrete classes must implement **all abstract methods**.

9. Interview / Industry Insight

- Interview Qs:
  - What is an abstract class?
  - Difference between abstract class and interface?
  - Why can't we instantiate abstract classes?
- Industry:
  - Abstract classes are used to enforce **coding contracts**.
  - Example: Django's `Model` class defines abstract methods to be implemented by subclasses.
  - Used when multiple subclasses **must share a common interface**.

4.11: Dataclasses (@dataclass)

1. Definition / Concept

- A **dataclass** is a Python class primarily used for **storing data** without writing boilerplate code.
- Introduced in **Python 3.7** via the `dataclasses` module.
- The `@dataclass` decorator automatically:
  - Adds an `__init__` method.
  - Adds `__repr__`, `__eq__`, and others.
- Makes code **cleaner** and **more readable** for classes that mostly hold data.

2. Analogy / Real-Life Connection

Think of a dataclass as a **form template**:

- Instead of writing the whole form from scratch (constructor, printing, comparison), the **dataclass auto-fills it for you**.
- You just declare the fields, and Python generates the boring code.

3. Syntax

```
from dataclasses import dataclass

@dataclass
class ClassName:
    field1: type
    field2: type
```

4. Step-by-Step Explanation

1. Import dataclass from dataclasses.
2. Use @dataclass decorator above your class.
3. Define fields with type hints (recommended).
4. Python automatically generates:
    - __init__()
    - __repr__() (for printing)
    - __eq__() (for comparison)

5. Example Code

(a) Simple Dataclass

```
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    age: int
    marks: float

s1 = Student("Alice", 20, 88.5)
print(s1)
```

Output:

```
Student(name='Alice', age=20, marks=88.5)
```

(b) Comparison with Normal Class

```
# Without dataclass
```

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

s1 = StudentNormal("Alice", 20)
s2 = StudentNormal("Alice", 20)
print(s1 == s2)  # False (compares memory addresses)
```

Output:

```
False
# With dataclass
@dataclass
class StudentData:
    name: str
    age: int

s1 = StudentData("Alice", 20)
s2 = StudentData("Alice", 20)
print(s1 == s2)  # True (compares values)
```

Output:

```
True
```

(c) Default Values in Dataclasses

```
@dataclass
class Course:
    name: str
    duration: int = 6  # default value

c1 = Course("Python")
print(c1)
```

Output:

```
Course(name='Python', duration=6)
```

(d) Using field() for Advanced Control

```
from dataclasses import dataclass, field
```

```
@dataclass
class Employee:
    name: str
    salary: float
    skills: list = field(default_factory=list)  # avoids mutable default

e1 = Employee("Bob", 50000)
e1.skills.append("Python")
print(e1)
```

Output:

```
Employee(name='Bob', salary=50000, skills=['Python'])
```

6. Diagram / Flow

📌 **What @dataclass Does Internally**

```
@dataclass
class Student:
    name: str
    age: int
```

Auto-generates:
- __init__(self, name, age)
- __repr__(self)
- __eq__(self)

7. Output

- Shown above.
- Dataclasses make printing and comparing objects more intuitive.

8. Common Errors & Debugging

❌ **Error 1: Forgetting to import dataclass**

```
@dataclass   # ❌ NameError
class Student:
```

name: str

✅ Fix: from dataclasses import dataclass

❌ **Error 2: Mutable default values**

@dataclass
class Test:
    values: list = [] # ❌ shared between objects

✅ Fix: Use field(default_factory=list)

❌ **Error 3: Using dataclasses without type hints**

- Works, but reduces readability and auto-features.

✅ Fix: Always add type hints (name: str).

9. Interview / Industry Insight

- Interview Qs:
  - What is a dataclass in Python?
  - Difference between normal class and dataclass?
  - Why use field(default_factory=...)?
- Industry:
  - Dataclasses used for **configurations, models, DTOs (Data Transfer Objects)**.
  - Often used in APIs, machine learning pipelines, and Django-like model replacements.
  - Cleaner than writing manual __init__, __repr__, __eq__.