1. Classes and Objects

- **Class**: A blueprint for creating objects (e.g., `Car`).
- **Object**: An instance of a class (e.g., `my_car`).

Example:

```
class Dog:
    # Class attribute (shared by all objects)
    species = "Canis familiaris"

    # Constructor (initializes object)
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age

    # Method (function inside a class)
    def bark(self):
        print(f"{self.name} says Woof!")
```

---

2. Creating Objects

```
# Create two Dog objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

print(dog1.name)   # Output: "Buddy"
print(dog2.species) # Output: "Canis familiaris"
dog1.bark()        # Output: "Buddy says Woof!"
```

---

3. The Four Pillars of OOP

a. Encapsulation

Bundling data (attributes) and methods into a single unit (class).
Example:

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

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

    def get_balance(self):
        return self.__balance
```

```
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

b. Inheritance

Creating a new class from an existing class.
Example:

```
class Animal:
    def speak(self):
        print("Animal sound")

class Cat(Animal):  # Inherits from Animal
    def speak(self):  # Method overriding
        print("Meow!")

cat = Cat()
cat.speak()  # Output: "Meow!"
```

c. Polymorphism

Using a single interface for different data types.
Example:

```
def animal_sound(animal):
    animal.speak()

dog = Dog("Buddy", 3)
cat = Cat()

animal_sound(dog)  # Output: "Buddy says Woof!"
animal_sound(cat)  # Output: "Meow!"
```

d. Abstraction

Hiding complex details and exposing only essential features.

```
from abc import ABC, abstractmethod

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

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

```
    def area(self):
        return 3.14 * self.radius ** 2

circle = Circle(5)
print(circle.area())  # Output: 78.5
```

---

4. Special Methods

Define how objects behave with operators (e.g., `+`, `==`).
Example:

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

    def __add__(self, other):  # Overloads '+'
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(2, 3)
v2 = Vector(1, 4)
result = v1 + v2
print(result.x, result.y)  # Output: 3 7
```

---

5. Key Terms

- **`self`**: Refers to the current instance of the class.
- **`__init__`**: Constructor method (called when an object is created).
- **Private Attributes**: Use `__` prefix (e.g., `__balance`).

Exercise

1. Create a `Book` class with `title`, `author`, and `pages` attributes.
2. Add a method `describe()` that prints:
   `"{title} by {author}, {pages} pages"`.
3. Create a subclass `Ebook` that inherits from `Book` and adds a `file_size` attribute.

---

1. What is OOP?

OOP is a programming paradigm based on the concept of **objects**, which contain **data** (attributes) and **functions** (methods) that operate on the data.

Key Concepts of OOP:

1. **Class** – Blueprint for creating objects
2. **Object** – Instance of a class
3. **Encapsulation** – Hiding internal details
4. **Inheritance** – Reusing code from a parent class
5. **Polymorphism** – Same function name behaving differently
6. **Abstraction** – Hiding complexity, showing essentials

---

2. Creating Classes and Objects

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

    def greet(self):
        print(f"Hi, I'm {self.name} and I'm {self.age} years old.")

# Creating an object
p1 = Person("Alice", 22)
p1.greet()  # Output: Hi, I'm Alice and I'm 22 years old.
```

---

3. `__init__` Method (Constructor)

This special method is automatically called when a new object is created.

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

---

4. Instance vs Class Variables

```
class Dog:
    species = "Canine"  # Class variable (shared)

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

d1 = Dog("Buddy")
```

```python
d2 = Dog("Charlie")

print(d1.species)  # Canine
print(d1.name)     # Buddy
```

---

5. Inheritance

A class can inherit methods and properties from another class.

```python
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):  # Inherits from Animal
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()  # Inherited method
d.bark()   # Own method
```

---

6. Method Overriding

Subclass can override a method from the parent class.

```python
class Animal:
    def speak(self):
        print("Animal sound")

class Cat(Animal):
    def speak(self):  # Overrides Animal's speak()
        print("Meow")

c = Cat()
c.speak()  # Output: Meow
```

---

7. Encapsulation

Using `_` or `__` to restrict access to variables/methods.

```python
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(1000)
acc.deposit(500)
print(acc.get_balance())  # 1500
# print(acc.__balance) -> Error (private)
```

---

8. Polymorphism

Same method name, different behavior depending on object.

```
class Bird:
    def sound(self):
        print("Tweet")

class Duck:
    def sound(self):
        print("Quack")

def make_sound(animal):
    animal.sound()

make_sound(Bird())  # Tweet
make_sound(Duck())  # Quack
```

---

9. `super()` Function

Used to call parent class methods in a child class.

```
class Animal:
    def __init__(self):
        print("Animal constructor")

class Dog(Animal):
    def __init__(self):
        super().__init__()  # Call parent constructor
        print("Dog constructor")

d = Dog()
```

```
# Output:
# Animal constructor
# Dog constructor
```

Summary Table

OOP Concept	Description
Class	Template for objects
Object	Instance of a class
Constructor	`__init__()` method
Inheritance	Derive new class from existing class
Encapsulation	Hide internal data/methods
Polymorphism	Same method, different behavior
Abstraction	Hiding details, exposing features