OOPS -- Object Oriented Programming System

These are the 4 pillars of OOPs in Python:
Encapsulation
Abstraction
Inheritance
Polymorphism

1. Real-World Modelling
OOP makes it easier to model real-world entities.
Example: A Car object can have attributes like brand, color, and behaviours like start(), stop().
This makes programs more natural and intuitive.

2. Code Reusability
Through inheritance, you can reuse code instead of writing it again and again.
Example: A Vehicle class can be reused for Car, Bike, Bus, etc.

3. Scalability & Maintainability
Large projects are easier to manage when broken into classes and objects.
If you need to update one part, you dont have to touch everything else.

4. Abstraction & Security
You can hide implementation details (abstraction) and protect data (encapsulation).
Example: A bank account class can hide the balance and only allow deposits/ withdrawals through methods.

5. Polymorphism (Flexibility)
Same method name works differently for different objects.
Example: sound() method → Dog barks, Cat meows.
This makes code more flexible and extensible.

6. To reduce redundancy

 # Procedural Programming

Step by step process
ex: step -1: To find next prime len>1
step -2: To do sum of digit of step 1 ans
step -3: Check ans of step 2 prime or not
Here code redundancy increases and no reusability

 # Functional Programming

Combining group of code in a method or function
ex: step -1: Function to check prime
step -2: To get next prime len>1
step -3: To get sum of digits of step 2 ans
step -4: TO check step 3 ans is prime or not
Here cannot recategorise group of functions

1. Class and Object: A class is a blueprint, and an object is an instance of that class.

```
# Defining a class
class Showroom:
    def __init__(self,brand,color):
        self.brand = brand
        self.color = color
        print(self.brand,self.color)

    def car_details(self):
        print(f"Brand: {self.brand} and Color: {self.color}")

#By creating object it automatically calls __init__
car1 = Showroom("Audi","Black")
car2 = Showroom("Rangerover","White")
car3 = Showroom("Tata","Orange")

#By using self it knows which car details you are asking
car1.car_details()
car3.car_details()
```

2. Inheritance: A class can inherit properties and methods from another class.

```
# Basic Inheritance: One class inherits from another.
# Parent class
class Animal:
    def speak(self):
        print("Animals make sounds")

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

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

Multilevel Inheritance: A class inherits from a child class (grandchild structure).

```
class Animal:
    def eat(self):
        print("Animal eats")

class Mammal(Animal):
    def walk(self):
        print("Mammal walks")

class Dog(Mammal):
    def bark(self):
        print("Dog barks")

# Object of lowest class
d = Dog()
d.eat()    # From Animal
d.walk()   # From Mammal
d.bark()   # From Dog
```

Multiple Inheritance: A class inherits from more than one parent class.

```
class Father:
    def f_skills(self):
        print("Father: Gardening and Programming")

class Mother:
    def m_skills(self):
        print("Mother: Cooking and Art")

class Child(Father, Mother):
    def skills(self):
        # Optionally call parent methods
        Father.f_skills(self)
        Mother.m_skills(self)
        print("Child: Music and Dancing")
        print()

c = Child()
c.skills()
c.f_skills()
c.m_skills()
```

Hierarchical Inheritance: Multiple child classes inherit from the same parent.

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

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

class Cat(Animal):
    def meow(self):
        print("Cat meows")

d = Dog()
c = Cat()
```

```
d.speak()
d.bark()
c.speak()
c.meow()
```

 # super() Keyword (Calling Parent Methods): Used to call a method from the parent class inside the child class.

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

class Dog(Animal):
    def __init__(self):
        super().__init__()  # Calls Animal constructor
        print("Dog created")
d = Dog()
```

Method Overriding: When a child class defines a method with the same name as in the parent class.

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

class Dog(Animal):
    def speak(self):  # Override
        print("Dog barks")

d = Dog()
d.speak()
```

3. Polymorphism: Same method name, different behaviour.

```
class Dog:
    def sound(self):
        return "Bark"

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

d = Dog()
c = Cat()

print(d.sound())
print(c.sound())

# Polymorphism in action
for animal in (Dog(), Cat()):
    print(animal.sound())
```

4. Encapsulation: Restricting access to variables/methods using _protected and __private.

Encapsulation means bundling data (variables) and methods (functions) that operate on that data into a single unit (class), and restricting direct access to some of the object's components.

Its mainly done using:
Public members — accessible everywhere
Protected members — accessible within the class and subclasses (_variable)
Private members — accessible only within the class (__variable)

```
# Example: Public, Protected, and Private Members

class Person:
    def __init__(self, name, age, salary):
        self.name = name          # Public
        self._age = age           # Protected
        self.__salary = salary    # Private

class Employee(Person):
    def show_details(self):
        # Accessing public and protected members in subclass
        print(f"Name: {self.name}")       # Public accessible
        print(f"Age: {self._age}")        # Protected accessible in subclass
        # print(self.__salary)            # Private not accessible in subclass

p = Person("Likhith", 21, 50000)
e = Employee("Kiran", 25, 60000)

# Public member — accessible everywhere
print(p.name)

# Protected member — can be accessed, but not recommended
print(p._age)        # Technically allowed

# Private member — not accessible directly
# print(p.__salary)   # AttributeError

# Access through subclass method
e.show_details()
```

5. Abstraction: Hiding details and showing only essential features (using abc module).

```
from abc import ABC, abstractmethod

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

class Circle(Shape):
    def __init__(self, r):
        self.r = r
    def area(self):
        return 3.14 * self.r * self.r

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