Scope of a Variables:-

1. Global Variables

Global variables are variables that are created **outside any function** (in the main program).

Key Points:

- Created **outside** the function
- Can be **accessed (read)** inside a function
- **Cannot be modified** inside a function directly
- To modify them inside a function, we must use the **global keyword**

Example 1: Accessing a Global Variable

```
x = 10   # global variable
def show_value():
    print("Inside function:", x)   # can read global variable

show_value()
print("Outside function:", x)
```

Output:

Inside function: 10
Outside function: 10

- **Global variable x is accessible inside and outside the function.**

Modifying a Global Variable

1. By default, if you **assign** a value inside a function, Python thinks it is a **local variable**.
2. To modify a global variable, use **global keyword**.

Example 2: Modifying Global Variable

```
count = 0   # global variable
def increment():
    global count
    count += 1

increment()
print(count)
```

```
print(count)
```

Output:	1

- global count tells Python to use the global variable.

Example 3: Modifying Without global

```
count = 0
def increment():
    count += 1   # Error: UnboundLocalError
increment()
print(count)
```

2. Local Variables

Local variables are variables created **inside a function**.

Key Points:

- Created **inside a function**
- Can be accessed and modified **only inside that function**
- Cannot be accessed outside the function

Example: Local Variable

```
def demo():
    a = 5   # local variable
    print(a)

demo()
# print(a)  # Error
```

HERE, a works inside the function but a does not work outside the function

Local Variable in Nested Function

If you want to modify a variable of the **outer function** inside an **inner function**, use **nonlocal keyword**.

Example: nonlocal Keyword

```python
def outer():
    x = 10
    def inner():
        nonlocal x
        x += 5
    inner()
    print(x)
outer()
```

Output:	15

Class and Object:

In programming:

- This **blueprint** is called a **Class**
- The real phones created from it are called **Objects**

Class:-

Class is a blueprint or template

- It stores **data (properties)**
- It defines **functionality (behavior)**

OR

Class is a container that holds properties and functions of real-time entities.

Example: Laptop

- **Laptop** → real-world entity → **Object**
- Laptop has:
  - Properties → brand, color, RAM, price
  - Functions → start(), shutdown(), charge()

All these properties and functions are **defined inside a class**.

Object:-

Object is an instance of a class

- From **one class**, we can create **many objects**

Example:

- Class → Mobile
- Objects → Samsung phone, iPhone, Redmi phone

Class Creation (Syntax):-

```
class ClassName:
    properties   # variables
    functions    # methods
```

Example:

```
class Mobile:
    brand = "Samsung"
    price = 20000
```

Object Creation:-

```
object_name = ClassName(arguments)
```

Example:

```
m1 = Mobile()
m2 = Mobile()
```

Here:

- Mobile → Class
- m1, m2 → Objects

Accessing Class Properties:-

Syntax:	object_name.property_name

Example:

```
print(m1.brand)
print(m2.brand)
```

```
print(m1.price)
```
Output:
```
Samsung
Samsung
20000
```
- Both objects access the **same class properties**.

Modifying / Updating Properties:-
You can update properties in **two ways**:

1. Using Class Name
```
ClassName.property_name = new_value
```
2. Using Object Name
```
object_name.property_name = new_value
```
⚠️ But the effect is **different**.

Example: Bank Class
```
class Bank:
    bname = "SBI"
    loc = "Hyderabad"
c1 = Bank()
c2 = Bank()
# Accessing Properties
print(Bank.bname, Bank.loc)
print(c1.bname, c1.loc)
print(c2.bname, c2.loc)
```
Output:
```
SBI Hyderabad
SBI Hyderabad
SBI Hyderabad
```

Modification Using Class Name
```
Bank.loc = "Bengaluru"
```

 # After Modification:

```
print(Bank.bname, Bank.loc)
print(c1.bname, c1.loc)
print(c2.bname, c2.loc)
```

Output:

```
SBI Bengaluru
SBI Bengaluru
SBI Bengaluru
```

*Note: Modification in class affects all objects of that class.

Modification Using Object Name

```
c1.loc = "Delhi"
```

 # After Object Update:

```
print(c1.bname, c1.loc)
print(c2.bname, c2.loc)
print(Bank.bname, Bank.loc)
```

Output:

```
SBI Delhi
SBI Bengaluru
SBI Bengaluru
```

⚠️ Modification in an object does NOT affect the class or other objects.

Types of States / Properties:-

There are **two types of properties** in a class:

1. **Static / Generic / Class Members**
2. **Specific / Object Members**

1. Static / Generic / Class Members

Static members are properties that are **common for all objects** of a class.

Example (School):

- School name
- Location

- Website
- Timing
- Contact number
- Uniform

*These values are **same for every student**, so they are called **Static Members**.

Code Example: Static Members

```
class School:
    sname = "ABC School"
    loc = "Prayagraj"
    website = "www.abcschool.com"
    timing = "9 AM - 3 PM"
```

2. Specific / Object Members

Specific members are properties that are **different for each object**.
Example (Student):

- Student name
- Roll number
- Student ID
- Age
- Class
- Phone number
- Blood group

* These values change **from student to student**, so they are called **Specific Members**.

Code Example: Specific Members

```
st1 = School()
st1.name = "Aditya"
st1.sid = 1
st1.age = 14
st1.cls = 5
st2 = School()
```

st2.name = "Adil"
st2.sid = 2
st2.age = 15
st2.cls = 6

Accessing Static vs Specific Members

print(st1.sname) # Static member
print(st1.name) # Specific member

Question:

Create a **Bank class**

- It should have **4 static (class) members**
- Create **3 objects**
- Each object should have **3 object (specific) members**

Step 1:

```
class Bank:
    bank_name = "SBI"
    branch = "Prayagraj"
    ifsc = "SBIN000123"
    country = "India"

c1 = Bank()
c1.name = "Aditya"
c1.age = 22
c1.phone = 9876543210
c1.pan = "ABCDE1234F"
c1.balance = 50000

c2 = Bank()
c2.name = "Rahul"
c2.age = 23
c2.phone = 9123456780
c2.pan = "BCDEF2345G"
c2.balance = 60000
```

```
c3 = Bank()
c3.name = "Amit"
c3.age = 24
c3.phone = 9988776655
c3.pan = "CDEFG3456H"
c3.balance = 45000

c4 = Bank()
c4.name = "Neha"
c4.age = 21
c4.phone = 9090909090
c4.pan = "DEFGH4567I"
c4.balance = 70000

c5 = Bank()
c5.name = "Priya"
c5.age = 22
c5.phone = 9012345678
c5.pan = "EFGHI5678J"
c5.balance = 80000
```

***this is NOT a good approach,**

- Too many lines
- Code becomes lengthy
- Not industry oriented
- Repeated work for every object

Step 2: Solution → Use a Function inside the Class

To reduce lines and repetition, we use a **function inside the class** so that:

- Same function can be used for all objects
- Code becomes clean and reusable

Methods:-

The function which are written inside the class are called as Methods.

There are 3 types of methods:-

1. Object / Instance Method
2. Class Method
3. Static Method

1. Object Method:-

1. Methods that **work on object data**
2. They **must use self**
3. Used to **access or modify object members**

Example:-

```
class Bank:
    # Static / Class Members
    bname = "SBI"
    branch = "Hyderabad"
    ifsc = "SBIN000123"
    helpline = "1800-11-2211"

    # Constructor (Object members)
    def __init__(self, name, age, phone, pan, balance):
        self.name = name
        self.age = age
        self.phone = phone
        self.pan = pan
        self.balance = balance

    # Object Method
    def display(self):
        print(self.name, self.phone, self.balance)

    def change_phone(self, new_phone):
        self.phone = new_phone
```

```
# Object creation
obj1 = Bank("Aditya", 22, "9876543210", "ABCDE1234F", 50000)

# Output
obj1.display()
obj1.withdraw(10000)
obj1.display()
obj1.change_phone("9999999999")
obj1.display()
```

Note*: Object methods are used to access or modify object-specific data and use self.

2. Class Method:-

1. Used to **work with class (static) members**
2. Uses **@classmethod decorator**
3. Uses **cls** instead of self

Example:-

```
class Bank:
    # Static / Class Members
    bname = "SBI"
    branch = "Hyderabad"
    ifsc = "SBIN000123"
    helpline = "1800-11-2211"

    # Constructor (Object members)
    def __init__(self, name, age, phone, pan, balance):
        self.name = name
        self.age = age
        self.phone = phone
        self.pan = pan
        self.balance = balance
```

```
    # Object Method
    def display(self):
        print(self.name, self.phone, self.balance)

    # Class Method - display bank details
    @classmethod
    def bank_details(cls):
        print(cls.bname, cls.branch, cls.ifsc, cls.helpline)

    # Class Method - change helpline number
    @classmethod
    def change_helpline(cls, new_number):
        cls.helpline = new_number


Bank.bank_details()
Bank.change_helpline("1800-00-0000")
Bank.bank_details()
```

2. Static Method:-

(NO static method - Utility logic kept outside the class)

```
def is_valid_amount(amount):
    return amount > 0

class Bank:
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
    def deposit(self, amount):
        if is_valid_amount(amount):   # using external function
            self.balance += amount
            print("Deposit successful:", self.balance)
```

```
        else:
            print("Invalid amount")

acc = Bank("Aditya", 5000)
acc.deposit(-100) ----> Invalid amount
```

WHY THIS IS A PROBLEM?

i. is_valid_amount belongs to Bank logic
ii. But it is outside the class

Trying OBJECT METHOD:-

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

    def is_valid_amount(self, amount):
        return amount > 0

    def deposit(self, amount):
        # must use self to call object method otherwise ERROR
        if self.is_valid_amount(amount):
            self.balance += amount
            print("Deposit successful:", self.balance)
        else:
            print("Invalid amount")


acc = Bank("Aditya", 5000)

acc.deposit(1000)  -----> Deposit successful: 6000
acc.deposit(-500)  -----> Invalid amount
```

PROBLEMS : -

i. cls is not used
ii. No class (static) data involved
iii. Misuses the purpose of @classmethod

Note*:	Correct output ≠ Correct design

WHY STATIC METHOD IS NEEDED?

i. To keep related utility logic inside the class
ii. WITHOUT using self or cls

```
class Bank:
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
    @staticmethod
    def is_valid_amount(amount):
        return amount > 0
    def deposit(self, amount):
        if Bank.is_valid_amount(amount):
            self.balance += amount
            print("Deposit successful:", self.balance)
        else:
            print("Invalid amount")

acc = Bank("Aditya", 5000)

acc.deposit(1000)   # ---> Deposit successful: 6000
acc.deposit(-500)   # Invalid amount
```

Note*:	Static methods are used to keep utility logic related to a class inside the class without forcing object or class dependency

Method Type	Uses	Should be used when
Object method	Self	Object data is used
Class method	Cls	Class data is used
Static method	Nothing	Only utility logic

4 Pillars of OOPS (Object Oriented Programming)

1. Inheritance
2. Abstraction
3. Encapsulation
4. Polymorphism

1. INHERITANCE

Inheritance allows **one class to acquire the properties and methods of another class.**
The class whose properties are inherited is called **Parent / Base class.**
The class that inherits is called **Child / Derived class.**

Why Inheritance?

- Code reusability
- Avoid duplication
- Easy maintenance