

Simple and Precise

STUDY MATERIAL *FOR CLASS 12TH* COMPUTER SCIENCE



BY SAJAD AKBAR RATHER

As per latest JKBOSE syllabus-2025



About the Author

Mr. Sajad Akbar Rather is a dedicated and passionate teacher with a strong academic background, holding a Master's Degree in Computer Applications (MCA) and a Master's in Education (M.Ed.). With a deep commitment to quality education and digital empowerment, he has been actively involved in promoting Computer Science and ICT education in Jammu and Kashmir.

Mr. Rather was honoured with the prestigious National Award for innovative Practices, by NCERT in 2018 and later received the UT-Level Best Teacher Award in 2022, recognizing his exceptional contribution to the field of digital learning and innovative classroom practices.

He serves as a Master Trainer and Key Resource Person (KRP) in ICT for the JK SCERT, where he regularly participated in workshops and training programs to empower teachers with 21st-century digital teaching tools. He is also the author of the textbook "Informatics Practices – Class 12", which has helped hundreds of students across the region understand complex concepts with ease.

Currently, Mr. Rather is teaching Computer Science, Coding, and Science at Government Boys Higher Secondary School, Bomai Baramulla, where he continues to inspire students through hands-on learning, coding projects, and digital innovation.

Preface

This book on Computer Science for Class 12 has been carefully prepared in accordance with the latest JKBOSE syllabus, with the primary aim of making the subject easier, clearer, and more meaningful for students.

Computer Science, though technical in nature, can become highly interesting and approachable when taught with simple language, relatable examples, and practical illustrations. Keeping this in mind, every topic in this book is presented in a step-by-step manner, using easy explanations, solved programs, diagrams, and exercises designed to strengthen core concepts.

Special emphasis has been placed on helping students not just memorize, but truly understand the fundamentals of programming, data handling, networking, and problem-solving using Python. The book also aims to develop logical thinking and digital skills necessary for success in both academics and future careers.

I sincerely hope that this book serves as a useful learning tool for students and a helpful resource for teachers. Suggestions and feedback for further improvement are most welcome.

**With warm regards and best wishes,
Sajad Akbar Rather**

Contents		
S.NO	Name of the Unit	Page Nos.
1	Control Structures in Python	1-9
2	User Defined Functions in Python	10-18
3	Introduction to Object Oriented Programming in Python	19-41
4	Data Structures in Python	42-54
5	Database and SQL	55-67
6	Boolean Logic	68-77
7	Networking and Cyber Security	78-86

Unit 1: Control Structures in Python

Control structures determine the flow of a program's execution. They allow programmers to decide which blocks of code to execute, repeat, or skip based on certain conditions or patterns.

There are **three main types of control structures in Python**:

Conditional Statements

Looping Statements

Jump Statements

1. Conditional Statements (Decision-Making)

These allow you to execute specific blocks of code depending on whether a condition is True or False.

a) if Statement

The most basic decision-making statement.

Executes a block only if the condition is true.

```
age = 20
if age >= 18:
    print("You are eligible to vote")
```

b) if-else Statement

Executes one block if the condition is true, and another if false.

```
age = 16
if age >= 18:
    print("Eligible to vote")
else:
    print("Not eligible to vote")
```

c) if-elif-else Statement

Checks multiple conditions in sequence.

As soon as one condition is True, its block is executed, and others are skipped.

```
Percentage = int(input("Enter Percentage here: "))
```

```
if Percentage >= 90:
```

```
    print("Grade A")
```

```
elif Percentage >= 75:
```

```
    print("Grade B")
```

```
elif Percentage >= 60:
```

```
    print("Grade C")
```

```
else:
```

```
    print("Fail")
```

2. Looping Statements (Iteration)

Loops are used when you want to execute a block of code multiple times.

a) for Loop

Iterates over a sequence like a list, string, or range.

Loops allow a block of code to be executed multiple times based on a condition or sequence.

Syntax:

```
for variable in sequence:
```

```
    # loop body
```

Programme

```
for i in range(1, 6):
```

```
    print(i)
```

Output : 1 2 3 4 5

Programme

```
for char in "Python":
```

```
    print(char)
```

output : P y t h o n

b) while Loop

Repeats as long as a condition remains true.

Syntax:

while condition:

 # loop body

Programme

```
count = 1
```

```
while count <= 5:
```

```
    print("Count is", count)
```

```
    count =count +1
```

output:

Count is 1

Count is 2

Count is 3

Count is 4

Count is 5

Finite Loop in Python

A finite loop is a loop that runs a specific number of times. It stops when the condition is no longer true or when the sequence ends.

Programme

```
for i in range(1, 6):
```

```
    print(i)
```

output : 1 2 3 4 5

- This loop executes 5 times.
- The range(1, 6) generates numbers from 1 to 5.
- After 5 iterations, the loop terminates automatically.

Infinite Loop in Python

An infinite loop is a loop that never ends unless:

You use a break statement to exit manually.

Or an external force stops the program.

Programme

```
while True:
```

```
    print("This will run forever!")
```

The condition True is always true.

This loop never ends unless interrupted manually (Ctrl+C) or a break is added.

Programme

```
while True:
```

```
    name = input("Enter your name (type 'exit' to stop): ")
```

```
    if name == 'exit':
```

```
        break
```

```
    print("Hello", name)
```

This will keep asking for names until the user types "exit".

Another Programme of infinite loop.

```
i = 0
```

```
while i >= 0:
```

```
    print(i)
```

```
    i = i+1
```

Here the condition remains always true

Nested for Loop in Python

A nested loop means a loop inside another loop. In a nested for loop:

The outer loop runs once.

Then the inner loop runs completely for each iteration of the outer loop.

Programme

```
fruits = ["mango", "apple", "grapes", "cherry"]
colors = ["red", "green", "pink"]
for x in fruits: // outer loop
    for y in colors: // inner loop
        print(x, y)
```

Output:

```
mango red
mango green
mango pink
apple red
apple green
apple pink
grapes red
grapes green
grapes pink
cherry red
cherry green
cherry pink
```

3. Jump Statements

Jump statements alter the normal flow of control in loops and conditional structures.

a) break

Immediately exits the loop, even if the condition is still true.

Programme

```
for i in range(10):
```

```
    if i == 5:
```

```
        break
```

```
    print(i)
```

output : 1 2 3 4

b) continue

Skips the current iteration and continues with the next one.

Programme

```
for i in range(5):
```

```
    if i == 2:
```

```
        continue
```

```
    print(i)
```

output : 0 1 3 4

Some Extra Programmes

Check if a Number is Even or Odd

```
num = int(input("Enter a number: "))
```

```
if num % 2 == 0:
```

```
    print("The number is Even")
```

```
else:
```

```
    print("The number is Odd")
```

Grade Card for 5 Subjects

```
print("Enter marks for 5 subjects (out of 100):")
```

```
s1 = float(input("Subject 1: "))
```

```
s2 = float(input("Subject 2: "))
```

```
s3 = float(input("Subject 3: "))
```

```
s4 = float(input("Subject 4: "))
```

```
s5 = float(input("Subject 5: "))
total = s1 + s2 + s3 + s4 + s5
percentage = total / 5
print("Total Marks:", total, "/ 500")
print("Percentage:", round(percentage, 2), "%")
if percentage >= 90:
    print("Grade: A+")
elif percentage >= 80:
    print("Grade: A")
elif percentage >= 70:
    print("Grade: B")
elif percentage >= 60:
    print("Grade: C")
elif percentage >= 50:
    print("Grade: D")
else:
    print("Grade: F (Fail)")
```

Temperature Conversion Program

```
choice = input("C to F (1) or F to C (2)? ")
if choice == "1":
    c = float(input("Celsius: "))
    f = c * 9/5 + 32
    print(f"{c}°C = {f:.1f}°F")
elif choice == "2":
    f = float(input("Fahrenheit: "))
    c = (f - 32) * 5/9
```

```
print(f"{f}°F = {c:.1f}°C")
else:
    print("Invalid choice")
```

Program to Check Prime Number:

```
num = int(input("Enter a number: "))
if num > 1:
    for i in range(2, num):
        if num % i == 0:
            print(f"{num} is not a prime number.")
            break
    else:
        print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

Fibonacci Series (Using If-Else)

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. Here's the Fibonacci sequence for the first few terms:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```
n = int(input("Enter the number of terms: "))
a, b = 0, 1
if n <= 0:
    print("Please enter a positive integer.")
elif n == 1:
    print("Fibonacci Series: 0")
else:
```

```
print("Fibonacci Series:")
print(a, b, end=" ")
for _ in range(2, n):
    next_term = a + b
    print(next_term, end=" ")
    a, b = b, next_term
```

Output

Enter the number of terms: 5

Fibonacci Series:

0 1 1 2 3

Factorial of a Number

```
num = int(input("Enter a number: "))
if num < 0:
    print("Factorial is not defined for negative numbers.")
elif num == 0:
    print("Factorial of 0 is 1")
else:
    fact = 1
    for i in range(1, num + 1):
        fact = fact * i
    print("Factorial of", num, "is", fact)
```

Output:

Enter a number: 3

Factorial of 3 is 6

Unit 2: User Defined Functions in Python.

What is a Function?

A function is a block of organized, reusable code that performs a specific task. Functions help divide a program into smaller and manageable parts.

Advantages of Using Functions

- Code Reusability
- Better Organization
- Easy to Debug and Maintain
- Promotes Modular Programming

Types of Functions in Python

Built-in Functions

Examples: print(), len(), type(), range(), etc.

User Defined Functions

Functions created by users to perform specific tasks as per their requirements.

User Defined Functions

Syntax:

```
def function_name(parameters):  
    """docstring (optional)"""  
    # body of the function  
    return [expression]
```

Example:

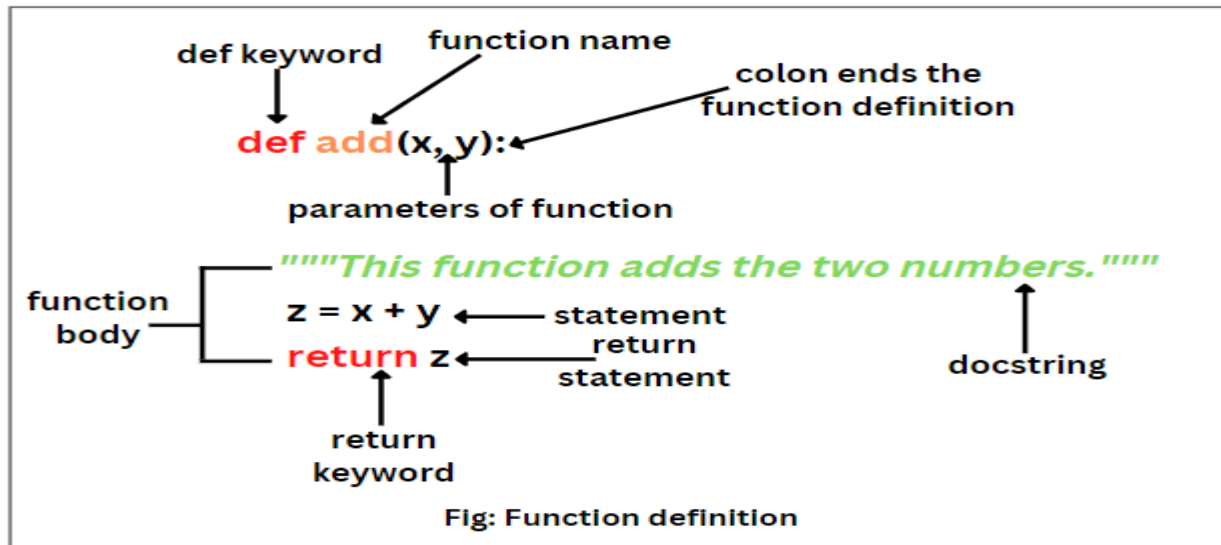
```
def greet(name):
```

```
    print("Hello", name)
```

```
greet("Sajad")
```

output : Hello Sajad

Components of a Function



Important Points

- Function name should follow identifier naming rules.
- Function must be defined before it is called.
- Use docstrings to describe what the function does.
- Avoid using the same names for local and global variables to reduce confusion.

Function with Return Value

```
def add(a, b):
```

```
    return a + b
```

```
result = add(10, 5)
```

```
print("Sum is:", result)
```

Output: Sum is: 15

Scope of Variables

Local Variables: Declared inside a function. Accessible only inside the function.

Global Variables: Declared outside all functions. Accessible throughout the program.

```
x = 10 # global  
  
def show():  
    x = 5 # local  
    print("Inside function:", x)  
  
show()  
  
print("Outside function:", x)
```

Output:

Inside function: 5

Outside function: 10

WHAT ARE ARGUMENTS?

When you define a function in Python, arguments are the values you pass to the function when calling it. These values are received by parameters in the function definition.

```
def greet(name, age):  
    print(f"Hello {name}, you are {age} years old.")  
  
greet("Sajad", 30)
```

"Sajad" and 30 are arguments

name and age are parameters

Simple Arguments (Regular Arguments)

These are the basic values you give to a function when you call it. You know exactly how many values you want to pass.

Example:

```
def greet(name, age):  
    print(f"Hello {name}, you are {age} years old.")  
  
greet("Sajad", 30)
```

Here, "Sajad" and 30 are arguments.

The function expects two values: name and age

Arbitrary Arguments

These are used when you don't know how many values you'll pass to the function.

a) **args* → *Arbitrary Positional Arguments*

Use **args* when you want to pass many values (like a list).

Inside the function, *args* acts like a tuple.

```
def add_numbers(*args):  
    total = 0  
    for num in args:  
        total = total + num  
    print("Total:", total)  
add_numbers(10, 20, 30)
```

Output: Total: 60

b) *kwargs* → *Arbitrary Keyword Arguments***

Use ***kwargs* when you want to pass many key=value pairs.

Inside the function, *kwargs* acts like a dictionary.

Example

```
def student_info(**details):  
    print(details)  
student_info(name="Ali", age=12)
```

Output: Total: 60

```
{'name': 'Ali', 'age': 12}
```

Here ***details* collects the keyword arguments (name="Ali", age=12) into a dictionary.

Another Programme to Print One by One

```
def student_info(**details):  
    for key, value in details.items():
```

```
print(key, ":", value)
student_info(name="Ali", age=12)
```

Output

```
name : Ali
age : 12
```

Recursive Function

A recursive function in Python is a function that calls itself to solve a problem by breaking it into smaller sub-problems of the same type. It's often used when a task can be divided into similar subtasks.

Recursive Function Example:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
```

Output: 120

Function that Returning a value and calling a function

A function that returns a value gives back a result when it finishes running.

we use the return keyword to send the result back to where the function was called.

```
def add(a, b):
    result = a + b
    return result      # this sends the answer back
sum = add(5, 3)        # calling the function
print("The sum is:", sum)
```

output :

The sum is: 8

How It Works:

add(5, 3) runs the function.

It adds 5 + 3 and stores it in result.

return result gives the value 8 back to the caller.

We store it in sum and print it.

Note:

If you don't use return, the function does its job but gives nothing back.

With return, you can use the value in other parts of your program.

```
def square(x):
```

```
    return x * x
```

```
print("Square is:", square(4))
```

```
# Output: Square is: 16
```

Some Programmes using functions.

Check Number is Even or Odd using return function

```
def check_even(num):
```

```
    if num % 2 == 0:
```

```
        return "Even"
```

```
    else:
```

```
        return "Odd"
```

```
result = check_even(7)
```

```
print("The number is:", result)
```

output

The number is: Odd

Find Maximum of Two Numbers

```
def find_max(a, b):
```

```
    if a > b:
```

```
    return a
else:
    return b
```

```
max_num = find_max(10, 25)
print("Maximum number is:", max_num)
```

output:

Maximum number is: 25

Convert Celsius to Fahrenheit

```
def c_to_f(celsius):
    return (celsius * 9/5) + 32
print("Temperature in Fahrenheit:", c_to_f(25))
```

output

Temperature in Fahrenheit: 77.0

Write a function that takes two numbers and returns the greater one.

```
def find_greater(num1, num2):
    if num1 > num2:
        return num1
    else:
        return num2
```

Example usage

```
a = 15
```

```
b = 27
```

```
greater = find_greater(a, b)
print("The greater number is:", greater)
```

Output : The greater number is: 27

Write a recursive function to calculate the sum of first n natural numbers.

```
def sum_natural(n):  
    if n == 1:  
        return 1  
    else:  
        return n + sum_natural(n - 1)  
  
# Example usage  
  
n = 5  
  
print("Sum of first", n, "natural numbers is:", sum_natural(n))
```

output:

Sum of first 5 natural numbers is: 15

Create a function that returns the square and cube of a number using return statement.

```
def square_and_cube(num):  
    square = num ** 2  
    cube = num ** 3  
    return square, cube  
  
# Example usage  
  
n = 4  
  
sq, cb = square_and_cube(n)  
print("Square of", n, "is:", sq)  
print("Cube of", n, "is:", cb)
```

output:

Square of 4 is: 16

Cube of 4 is: 64

Calculate Area of Circle

```
def area_circle(radius):  
    pi = 3.1416  
    return pi * radius * radius
```

```
print("Area is:", area_circle(5))
```

Output :

Area is: 78.54

Calculate Simple Interest

```
def simple_interest(p, r, t):  
    return (p * r * t) / 100  
  
interest = simple_interest(1000, 5, 2)  
print("Simple Interest is:", interest)
```

Output:

Simple Interest is: 100.0

Unit 3: Basic Concept of Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a method of programming where we structure software as a collection of objects, which combine data and functions together.

Instead of just writing procedures (like in traditional programming), OOP focuses on creating models based on real-world things.

Four Basic Concepts of OOP

Class

A class is a blueprint for creating objects.

It defines properties (variables) and behaviours (methods/functions).

Example: Car, Student, Dog.

Object

An object is a real instance of a class.

It contains actual values and can perform actions defined in the class.

Example: `myCar = Car("Red", "Toyota")`.

Encapsulation

Hides the internal details of objects.

Keeps data safe and secure by restricting direct access.

Uses private variables and public methods (getters/setters).

Inheritance

Allows a class to inherit properties and methods from another class.

Promotes code reuse.

Example: Dog inherits from Animal.

Polymorphism

Means “many forms.”

Allows the same method to behave differently for different objects.

Example: A method `speak()` can make a dog bark and a cat meow.

Advantages of Object-Oriented Programming (OOP)

1. Modularity

Code is organized into classes and objects, making it modular.

Each object can be developed, tested, and debugged independently.

2. Reusability

Using inheritance, existing classes can be reused to create new classes.

Saves time and effort in writing repetitive code.

3. Data Security (Encapsulation)

Data is protected by restricting access using private or protected attributes.

Only specific methods can access or modify the data.

4. Flexibility and Maintainability

Changes in one part of the program have minimal impact on others.

Makes software easier to maintain and update.

5. Polymorphism

A single function or method can work in different ways depending on the object.

Increases flexibility and reduces complexity in code.

6. Real-World Modeling

OOP models real-world entities more naturally using classes and objects.

Makes understanding and designing software more intuitive.

7. Ease of Testing and Debugging

Code is divided into objects, so it's easier to test and debug specific components.

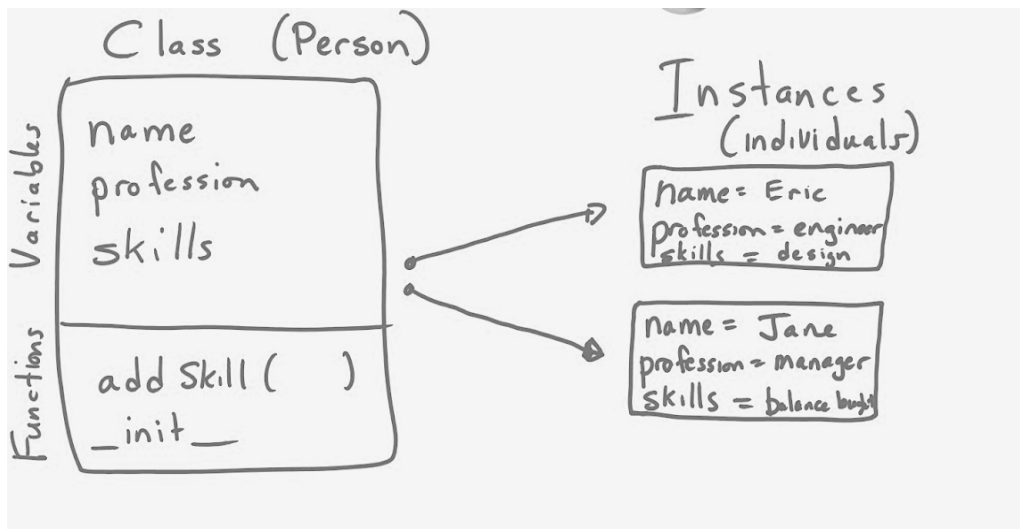
8. Extensibility

New features or behaviors can be easily added by extending existing classes.

What is a Class?

A class is a blueprint or template used to create objects in Object-Oriented Programming (OOP).

It defines a structure that describes properties (data/variables) and behaviors (methods/functions) that the objects created from the class will have.



Create a class named Employee

class Employee:

This function runs when we create a new employee

def __init__(self, name, salary):

self.name = name # name of the employee

self.salary = salary # salary of the employee

This function displays employee details

def show_details(self):

print("Name:", self.name)

print("Salary:", self.salary)

(Creating an Object)

Create an employee object

emp1 = Employee("Ali", 30000)

Call the function to show details

emp1.show_details()

output:

Name: Ali

Salary: 30000

What is an Object?

An object is a real instance of a class.

When you create an object from a class, you can use its data and functions.

Class = Blueprint

Object = Real Thing Created Using the Blueprint

An object contains:

Data (Attributes or Properties) → e.g., name, age, salary

Behavior (Methods or Functions) → e.g., speak(), drive(), show_info()

Multiple objects can be created from the same class.

Each object has its own copy of data but can use the same methods.

Step 1: Define the Class

class Student:

```
def __init__(self, name, marks):  
    self.name = name  
    self.marks = marks  
  
def display_result(self):  
    print("Student Name:", self.name)  
    print("Marks:", self.marks)  
    if self.marks >= 33:  
        print("Result: Pass")  
    else:  
        print("Result: Fail")
```

Step 2: Create an Object and Use It

Creating a student object

```
student1 = Student("Areeba", 45)
```

Calling the method to display result

```
student1.display_result()
```

Output:

Student Name: Areeba

Marks: 45

Result: Pass

Another Example (Fail Case)

```
student2 = Student("Faizan", 28)
```

```
student2.display_result()
```

Output:

Student Name: Faizan

Marks: 28

Result: Fail

What is Encapsulation in OOP?

Encapsulation means hiding the internal details of an object and only exposing what is necessary.

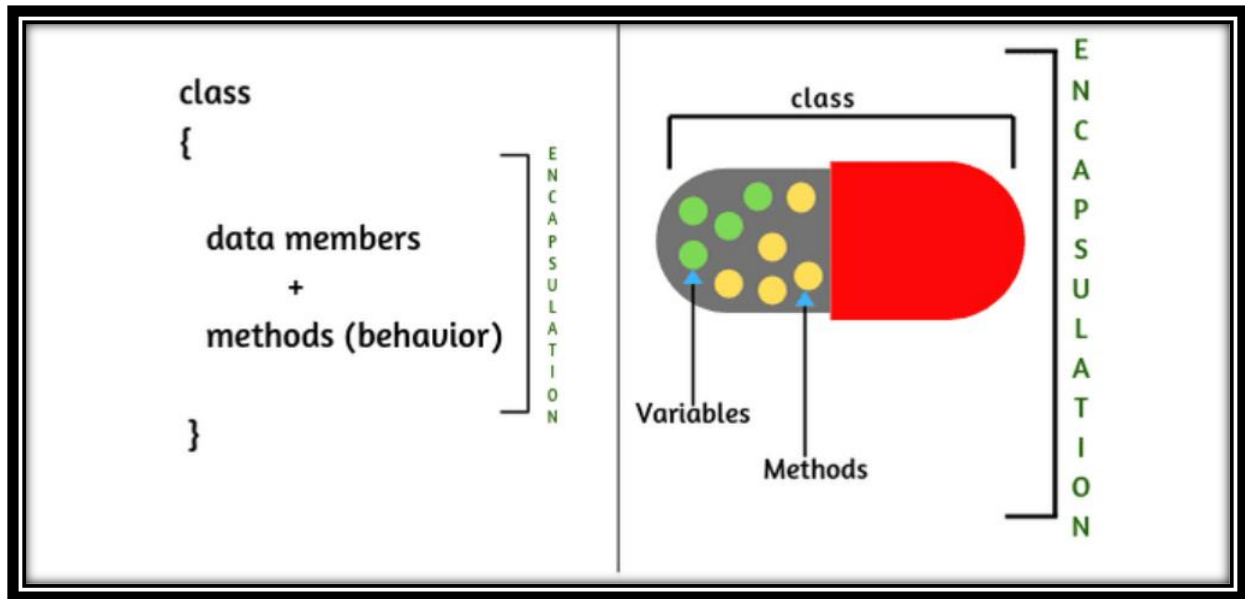
In other words:

It is the process of binding data (variables) and functions (methods) together in one unit – the class – and controlling access to the data and has following advantages

Data is hidden inside the class so that no one can change it directly.

We use special functions (called getters and setters) to see or change the data safely.

It protects data from being changed wrongly or by mistake.



```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.__age = age # private variable
```

```
    def get_age(self):
```

```
        return self.__age # getter method
```

```
# Create an object
```

```
p1 = Person("John", 25)
```

```
# Access data using method
```

```
print(p1.name)    # public attribute
```

```
print(p1.get_age()) # accessing private attribute via getter
```

Output:

```
John
```

```
25
```

Key Points:

name is public and can be accessed directly.

__age is private and accessed using the get_age() method.

Polymorphism in Python

Polymorphism means "many forms".

In programming, it allows the same method or function name to behave differently based on the object or data type.

Basic Advantages of Polymorphism :

Code Reusability

► You can write one function or method and use it for many different types of objects.

Simplicity and Clean Code

► Code becomes easier to read and understand because the same method name is used.

Easy to Maintain

► If changes are needed, you don't have to change code everywhere — only in one place.

Extensibility

► You can easily add new classes and features without changing existing code.

Flexibility

► You can use the same function with different objects, and it will work correctly based on the object type.

Programme

```
def add(a, b):  
    return a + b  
  
# Adding numbers  
print(add(10, 5))    # Output: 15  
  
# Adding strings  
print(add("Hello ", "World")) # Output: Hello World  
  
# Adding lists
```

```
print(add([1, 2], [3, 4]))    # Output: [1, 2, 3, 4]
```

Output:

15

Hello World

[1, 2, 3, 4]

Here same function add() works with:

integers (adds numbers),

strings (joins text),

lists (joins elements).

i.e Same function name gives different results based on the data type. That is called polymorphism.

What is Inheritance?

Inheritance is a concept in Object-Oriented Programming where:

A child class (also called subclass) inherits the properties and methods of a parent class (also called superclass).

It helps in reusing code and building relationships between classes. inheritance means one class can use the features of another class.

Advantages of Polymorphism in OOP:

Code Reusability

You can use the same method name in different classes.

This avoids writing the same code again and again.

Flexibility

The same method works differently based on the object calling it.

Easier Maintenance

Since code is reused and structured, it's easier to manage and update.

Improves Readability

Using the same function name across classes makes the code easier to understand.

Extensibility

You can add new classes or features without changing existing code much.

Inheritance Example: Addition and Subtraction

Parent Class: Calculator

Child Class: AdvancedCalculator (Inherits from Calculator)

Parent class

```
class Calculator:
```

```
    def add(self, a, b):
```

```
        return a + b
```

```
    def subtract(self, a, b):
```

```
        return a - b
```

Child class inherits from Calculator

```
class AdvancedCalculator(Calculator):
```

```
    def multiply(self, a, b):
```

```
        return a * b
```

```
    def divide(self, a, b):
```

```
        return a / b if b != 0 else "Cannot divide by zero."
```

Create object of AdvancedCalculator class

```
calc = AdvancedCalculator()
```

Using methods from both parent and child classes

```
print("Addition:", calc.add(10, 5))    # From Calculator
```

```
print("Subtraction:", calc.subtract(10, 5)) # From Calculator
```

```
print("Multiplication:", calc.multiply(10, 5)) # From AdvancedCalculator
```

```
print("Division:", calc.divide(10, 5))    # From AdvancedCalculator
```

Output:

Addition: 15

Subtraction: 5

Multiplication: 50

Division: 2.0

Explanation:

The AdvancedCalculator class inherits methods add() and subtract() from the parent class Calculator.

It adds its own methods multiply() and divide().

This allows us to use all four operations on the same object.

Types of inheritance

Single Inheritance:

In single inheritance, a child class inherits from one parent class.

class A:

```
def show(self):
```

```
    print("This is class A")
```

```
class B(A): # Class B inherits from class A
```

```
obj = B()
```

```
obj.show() # Inherited method from class A
```

✅ Output:

This is class A

Note:

In inheritance, it allows a child class to inherit from a parent class without adding or modifying anything.

2. Multiple Inheritance:

In multiple inheritance, a child class inherits from two or more parent classes.

python

class A:

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

```
class B:  
    def greet(self):  
        print("Hello from class B")  
class C(A, B): # Class C inherits from both A and B  
obj = C()  
obj.show() # Inherited from class A  
obj.greet() # Inherited from class B
```

Output:

This is class A

Hello from class B

3. Multilevel Inheritance:

In multilevel inheritance, a child class inherits from a parent class, and that parent class itself is inherited from another class.

```
class A:  
    def show(self):  
        print("This is class A")  
class B(A): # Class B inherits from A  
    def display(self):  
        print("This is class B")  
class C(B): # Class C inherits from B, which inherits from A  
    pass  
obj = C()  
obj.show() # Inherited from class A  
obj.display() # Inherited from class B
```

Output:

This is class A

This is class B

4. Hierarchical Inheritance:

In hierarchical inheritance, multiple child classes inherit from a single parent class.

python

class A:

def show(self):

print("This is class A")

class B(A): # Class B inherits from A

class C(A): # Class C also inherits from A

obj_b = B()

obj_b.show() # Inherited from class A

obj_c = C()

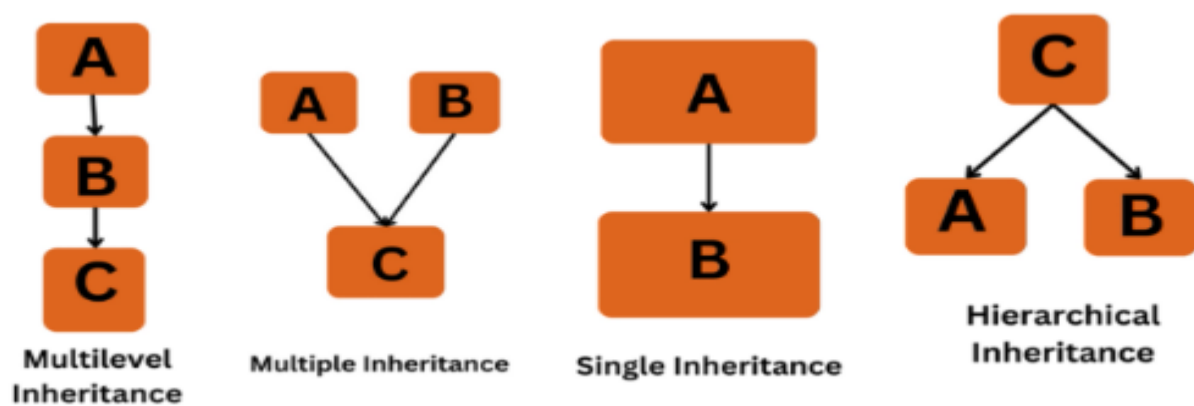
obj_c.show() # Inherited from class A

Output:

This is class A

This is class A

Summary



Inheritance Type	Description	Example
Single Inheritance	One child class inherits from one parent class.	B inherits from A
Multiple Inheritance	One child class inherits from two or more parent classes.	C inherits from both A and B
Multilevel Inheritance	A child class inherits from a parent class, and that parent inherits from another class.	C inherits from B, and B inherits from A
Hierarchical Inheritance	Multiple child classes inherit from a single parent class.	B and C both inherit from A

Data Abstraction

Definition:

Data Abstraction refers to hiding the internal implementation details of a program and showing only the essential features to the user.

Advantages of Data Abstraction:

- Only the necessary information is shown.
- Complex logic is hidden from the user.
- Achieved using abstract classes and methods in Python.
- Helps in reducing complexity and improving code clarity.

```
from abc import ABC, abstractmethod
```

```
class Operation(ABC):
```

```
    @abstractmethod
```

```
    def perform(self, a, b):
```

```
        pass
```

```
class Add(Operation):
```

```
    def perform(self, a, b):
```

```
        return a + b
```

```
class Subtract(Operation):
```

```
    def perform(self, a, b):
```

```
        return a - b
```

Using the classes

```
a, b = 10, 5
```

```
print("Addition:", Add().perform(a, b))
```

```
print("Subtraction:", Subtract().perform(a, b))
```

Lets understand this Programme.

```
from abc import ABC, abstractmethod
```

This line is **importing tools** from Python's abc module, which is used to create **abstract classes**.

Part	Meaning
Abc	Stands for Abstract Base Classes – a module in Python.
ABC	A base class used to create abstract classes (like a blueprint).
abstractmethod	A decorator that marks a method as abstract – meaning it must be defined in any subclass.

ABC = Abstract Base Class.

abstractmethod = lets us define a method that has no body (empty).

```
class Calculator(ABC): # Abstract class
```

Defines an abstract class called Calculator:

This is a general class that other classes will inherit from.

It cannot be used directly to create objects.

```
    @abstractmethod
```

```
    def calculate(self, a, b):
```

```
        pass
```

This is an abstract method:

It tells that every class that inherits from Calculator must define a calculate method.

It doesn't do anything here – the real code will be in the child classes.

```
class Addition(Calculator):
```

A child class called Addition that inherits from Calculator.

```
def calculate(self, a, b):  
    return a + b
```

This method is the real implementation of addition:

It overrides the abstract method from the parent class.

It returns the sum of a and b.

```
class Subtraction(Calculator):
```

Another child class, this time for subtraction.

```
def calculate(self, a, b):  
    return a - b
```

This method performs subtraction.

It returns the result of a - b.

```
add = Addition()
```

Creates an object add of class Addition.

```
sub = Subtraction()
```

Creates an object sub of class Subtraction.

```
print("Addition:", add.calculate(5, 3))
```

Calls the calculate() method of Addition class with values 5 and 3:

Output: Addition: 8

```
print("Subtraction:", sub.calculate(5, 3))
```

Calls the calculate() method of Subtraction class with values 5 and 3:

Output: Subtraction: 2

Data Hiding

Data Hiding means restricting access to the internal data of a class to prevent it from being changed directly.

Key Advantages:

- Data is made private using double underscores __.
- Access is given through getter and setter methods.

- Improves security and safety of data.

Example:

```
class Student:
```

```
    def __init__(self):
        self.__name = ""

    def set_name(self, n):
        self.__name = n

    def get_name(self):
        return self.__name
```

```
# Create object
```

```
s = Student()
```

```
s.set_name("Ali")
```

```
print("Name:", s.get_name())
```

Output:

Ali

Lets learn this Programme line by line

```
class Student:
```

```
    def __init__(self):
        self.__name = ""
```

```
class Student:
```

This defines a class named Student.

```
def __init__(self):
```

This is the constructor. It runs automatically when a new object is created.

```
self.__name = ""
```

__name is a private variable (because of the double underscore __).

It is hidden from direct access outside the class.

Initially, it is set to an empty string "".

```
def set_name(self, n):
```

```
    self.__name = n
```

```
def set_name(self, n):
```

This is a setter method used to safely assign a value to __name.

```
self.__name = n
```

The value of n (passed by the user) is stored in the private variable __name.

```
def get_name(self):
```

```
    return self.__name
```

```
def get_name(self):
```

This is a getter method used to safely read the value of __name.

```
return self.__name
```

It returns the stored name to the user.

```
# Create object
```

```
s = Student()
```

This creates an object s of the class Student.

It also calls the constructor (__init__), so __name is initialized as "".

```
s.set_name("Ali")
```

Calls the set_name() method and passes "Ali" to it.

"Ali" is stored in the private variable __name.

```
print("Name:", s.get_name())
```

Calls the get_name() method to retrieve the name.

Prints: Name: Ali

Difference between Data Abstraction and Data Hiding

Feature	Data Abstraction	Data Hiding
Focus	Hides implementation logic	Hides internal data/ attributes
Goal	Simplifies usage	Protects data
Achieved by	Abstract classes & methods	Private variables + getter/setter
User knows about	What the object does	Only controlled access to data

Access specifiers

In Python, access specifiers (also known as visibility modifiers) control the access level of variables and methods in a class. They help in managing the visibility of data and prevent unauthorized access or modification.

There are three types of access specifiers in Python:

Public

Protected

Private

1. Public Access Specifier:

Public members (variables or methods) are accessible from anywhere, inside or outside the class.

By default, all members (attributes and methods) in Python are public.

Example:

class Person:

```
    def __init__(self, name, age):  
        self.name = name # Public attribute  
        self.age = age   # Public attribute  
    def greet(self):     # Public method  
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

Creating an object

```
person = Person("John", 30)
```

Accessing public attribute and method

```
print(person.name) # Accessible from outside
```

```
person.greet()     # Accessible from outside
```

Output:

Hello, my name is John and I am 30 years old.

2. Protected Access Specifier:

Protected members are intended to be accessed only within the class and by subclasses (child classes).

In Python, protected members are indicated by a single underscore (_) before the variable or method name.

Note: This is just a convention; it does not restrict access, but it signals to the developer that these variables or methods should not be accessed directly from outside the class.

Example:

```
class Person:

    def __init__(self, name, age):

        self._name = name # Protected attribute

        self._age = age   # Protected attribute

    def _greet(self):      # Protected method

        print(f"Hello, my name is {self._name} and I am {self._age} years old.")

# Creating an object

person = Person("John", 30)

# Accessing protected attribute and method

print(person._name) # Not recommended, but possible

person._greet()     # Not recommended, but possible
```

Output:

John

Hello, my name is John and I am 30 years old.

Although you can access protected members, the underscore signifies that these are meant to be protected and should ideally not be accessed directly from outside the class.

3. Private Access Specifier:

Private members are intended to be accessed only within the class where they are defined.

In Python, private members are indicated by a double underscore (__) before the variable or method name.

Note: Like protected members, Python does not strictly prevent access to private members. However, it mangles the name internally to make it harder to accidentally access them from outside the class.

Example:

```
class Person:
```

```
    def __init__(self, name, age):  
        self.__name = name # Private attribute  
        self.__age = age   # Private attribute  
    def __greet(self): # Private method  
        print(f"Hello, my name is {self.__name} and I am {self.__age} years old.")
```

```
# Creating an object
```

```
person = Person("John", 30)
```

```
# Accessing private attribute and method
```

```
# print(person.__name) # This will raise an error because it's private
```

```
# person.__greet()    # This will raise an error because it's private
```

Output:

John

Hello, my name is John and I am 30 years old.

Summary

Access Specifier	Description	Convention	Access Example
Public	Members are accessible from anywhere, both inside and outside the class.	No leading underscore (name)	object.name
Protected	Members are intended to be accessed only inside the class and by subclasses (child classes).	One leading underscore (_name)	object._name
Private	Members are intended to be accessed only within the class itself, not from outside the class.	Two leading underscores (__name)	object.__name

Conclusion:

- Use public for attributes and methods that need to be accessed outside the class.
- Use protected as a signal that certain members are meant for internal use or subclass use.
- Use private for attributes and methods that should not be accessed outside the class, ensuring data encapsulation.

What is a Constructor?

A constructor is a special method in Python used to initialize objects of a class.

It is automatically called when an object is created.

The name of the constructor method is always `__init__()`.

Types of Constructors:

- **Default Constructor**
- **Parameterized Constructor**

1. Default Constructor

A constructor with no parameters (except self).

Used when we don't need to give any data during object creation.

Example:

class Student:

```
def __init__(self): # Default constructor  
    print("This is a default constructor")
```

s = Student() # Object created, constructor runs automatically

Output

This is a default constructor

2. Parameterized Constructor

A constructor that takes parameters to assign values to object variables.

Allows us to pass data when creating an object.

class Student:

```
def __init__(self, name, marks): # Parameterized constructor
```

```
self.name = name  
  
self.marks = marks  
  
def show(self):  
    print("Name:", self.name)  
    print("Marks:", self.marks)  
  
s = Student("Ali", 85)  
  
s.show()
```

Output:

Name: Ali

Marks: 85

Here in this programme :

`__init__()` is the constructor.

`self` refers to the current object.

We use `self.name = name` to store the passed value in the object.

When we create the object `s = Student("Ali", 85)`, the constructor is called with values "Ali" and 85.

Summary Table:

Type	Description	Example
Default Constructor	No parameters, runs automatically	<code>def __init__(self):</code>
Parameterized Constructor	Takes input values to initialize data	<code>def __init__(self, name, age):</code>

What is a Destructor?

A destructor is a special method in Python that is called automatically when an object is deleted or no longer in use.

Its purpose is to clean up resources (like closing files, releasing memory).

In Python, the destructor method is named: `__del__()`

When is it Called?

Automatically when:

- The object is deleted using del keyword
- The program ends and the object is destroyed
- The object goes out of scope

Example:

```
class Student:
```

```
    def __init__(self):
        print("Constructor is called")
    def __del__(self):
        print("Destructor is called")
```

Another Example:

Simple Destructor Example

```
class Car:
```

```
    def __init__(self, brand):
        self.brand = brand
print(f"{self.brand} car is created.")
    def __del__(self):
        print(f"{self.brand} car is destroyed.")
# Create object
my_car = Car("Toyota")
# Delete object
del my_car
```

Output: Toyota car is created. Toyota car is destroyed.

Here in this Programme:

The `__init__()` constructor runs when we create the object `my_car`.

The `__del__()` destructor runs when we delete the object using `del my_car`.

Python also automatically calls the destructor when the program ends or the object is no longer used.

Unit :4 Data structures in Python

Data structure is a way of organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently.

It acts like a container or format that helps the computer perform tasks such as storing, searching, sorting, and updating data in a logical and efficient way.

Advantages of Data Structures

- Efficient Data Handling – Organize and manage data easily.
- Faster Access – Retrieve and update data quickly.
- Less Memory Use – Save memory by choosing the right structure.
- Avoids Duplicates – Structures like sets store only unique values.
- Better Problem Solving – Helps write efficient programs and algorithms.

What is an Array?

An array is a collection of items stored at contiguous memory locations. It stores multiple values of the same data type under a single name.

Steps

1. Importing the Array Module
2. Creating an Array

Syntax:

```
array_name = array.array(typecode, [elements])
```

typecode specifies the type of elements (e.g., 'i' for integers).

[elements] is a list of values of the same type.

Example:

```
import array
```

```
marks = array.array('i', [45, 67, 89, 32])
```

This creates an integer array named marks.

Accessing Elements of an Array

We can access array elements like

```
print(marks[0]) # Output: 45    / / Indexing starts from 0.
```

```
print(marks[2]) # Output: 89
```

Length of an Array

To find how many elements are in the array, use:

```
print(len(marks))
```

Output: 4

Traversing (Looping through) an Array

we can use a for loop to display all elements as shown below:

```
import array
```

```
marks = array.array('i', [45, 67, 89, 32])
```

```
for m in marks:
```

```
    print(m)
```

Output:

45

67

89

32

What is a List?

A list is a built-in data structure in Python used to store a collection of items.

Lists are ordered, changeable (mutable), and allow duplicate values.

Creating a List

Example:

```
fruits = ["apple", "banana", "cherry"]
```

Accessing Elements in a List

We can access elements using index numbers, starting from 0.

```
print(fruits[0]) # Output: apple
```

```
print(fruits[1]) # Output: banana
```

Modifying List Elements

We can change any element in the list using its index.

```
fruits[1] = "mango"
```

```
print(fruits)
```

Output: ['apple', 'mango', 'cherry']

Method	Description	Example
append()	Adds item at the end	fruits.append("orange")
insert()	Adds item at a specific index	fruits.insert(1, "grapes")

```
fruits.append("orange")
```

```
print(fruits)
```

#Output ['apple', 'mango', 'cherry', 'orange']

Removing Elements from a List

Method	Description	Example
remove()	Removes a specific item	fruits.remove("apple")
pop()	Removes item at a specific index	fruits.pop(1)
del	Deletes item using index	del fruits[0]

Length of a List

```
print(len(fruits))
```

Output: 4

Looping Through a List

```
fruits = ["apple", "banana", "cherry"]
```

```
fruits.append("orange")
```

```
fruits[1] = "mango"
```

```
for fruit in fruits:
```

```
    print(fruit)
```

#output

apple

mango

cherry

orange

Types of Lists in Python:

Ordered:

The items in a list are stored in the order in which they are added. This means the position of elements is maintained.

Changable:

Lists are mutable, meaning you can change, add, or remove elements after the list is created.

Allows Duplicates:

Lists can have duplicate elements, meaning the same value can appear more than once in the list.

1. Ordered

Since lists are ordered, the order in which you insert elements is preserved. You can access elements using their index.

Example:

```
fruits = ["apple", "banana", "cherry"]
```

```
print(fruits[0]) # Output: apple
```

```
print(fruits[1]) # Output: banana
```

Here, fruits[0] will always give you "apple" because the list is ordered.

2. Changable

Lists are mutable, so you can change an element, add a new one, or remove an existing one.

Example:

```
fruits = ["apple", "banana", "cherry"]
```

```
fruits[1] = "orange" # Change 'banana' to 'orange'
```

```
print(fruits) # Output: ['apple', 'orange', 'cherry']
```

Adding an element

```
fruits.append("grape")
```

```
print(fruits) # Output: ['apple', 'orange', 'cherry', 'grape']
```

Removing an element

```
fruits.remove("cherry")
```

```
print(fruits) # Output: ['apple', 'orange', 'grape']
```

3.Allows Duplicates

Lists can contain duplicate values. The same item can appear multiple times.

Example:

```
fruits = ["apple", "banana", "apple", "cherry", "banana"]
```

```
print(fruits) # Output: ['apple', 'banana', 'apple', 'cherry', 'banana']
```

In this list, both "apple" and "banana" appear twice.

Dictionary in Python

A dictionary is a collection of key-value pairs. Each value is accessed using a unique key.

Syntax:

```
dict_name = {key1: value1, key2: value2, ...}
```

Types of Dictionary:

1. Ordered

Dictionaries maintain the insertion order of key-value pairs.

Example:

```
student = {"name": "Ali", "age": 17, "class": "12th"}
```

```
print(student)
```

Output:

```
{'name': 'Ali', 'age': 17, 'class': '12th'}
```

Items appear in the same order they were added.

2. Changeable (Mutable)

Dictionaries are mutable, which means you can change, add, or remove items.

```
student = {"name": "Ali", "age": 17}
```

```
# Modify value
```

```
student["age"] = 18
```

```
# Add new key-value pair
```

```
student["class"] = "12th"
```

```
print(student)
```

Output:

```
{'name': 'Ali', 'age': 18, 'class': '12th'}
```

3. No Duplicate Keys Allowed

Dictionaries do not allow duplicate keys.

If the same key is repeated, the last value is kept.

Example:

```
student = {"name": "Ali", "age": 17, "age": 18}
```

```
print(student)
```

```
# Output: {'name': 'Ali', 'age': 18}
```

Only the last value of "age": 18 is stored.

However, values can be duplicated.

Example:

```
marks = {"math": 95, "science": 95}
```

```
print(marks)
```

```
# Output: {'math': 95, 'science': 95}
```

Length of a Dictionary

Example:

```
# Creating a dictionary
student = {
    "name": "Ali",
    "age": 17,
    "class": "12th"
}

# Finding the length using len()
length = len(student)

# Display the result
print("The number of key-value pairs in the dictionary is:", length)
```

Output:

The number of key-value pairs in the dictionary is: 3

Explanation:

len(dictionary) returns the number of key-value pairs in the dictionary.

In this case, student has 3 pairs, so the output is 3

Accessing Dictionary Elements

Access elements using their key:

```
print(student["name"])
```

Output: Ali

Accessing a Dictionary using a for loop

A dictionary stores key-value pairs. We can use a for loop as given below:

1. Loop through keys

```
student = {"name": "Ali", "age": 17, "class": "12th"}
```

for key in student:

```
    print(key, ":", student[key])
```

Output:

name : Ali

age : 17

class : 12th

2. Loop through keys using .keys()

for key in student.keys():

 print(key)

Output:

name

age

class

3. Loop through values using .values()

for value in student.values():

 print(value)

Output:

Ali

17

12th

4. Loop through key-value pairs using .items()

for key, value in student.items():

 print(key, "=", value)

Output:

name = Ali

age = 17

class = 12th

Types of Dictionary (Summary)

Feature	Description
Ordered	Maintains insertion order (Python 3.7+)
Changeable	You can add, update, or remove items
No Duplicate Keys	Keys must be unique; latest value is kept if duplicate keys are used

Duplicates in Values	Allowed – same value can appear multiple times under different keys
----------------------	---

Tuples in Python

A tuple is a collection data type in Python that is:

- Ordered: Items have a defined order.
- Unchangeable (Immutable): Items cannot be changed after the tuple is created.
- Allows Duplicates: Duplicate values are allowed.

Defined using parentheses ().

Creating a Tuple

Basic Syntax:

```
my_tuple = (item1, item2, item3)
```

Example:

```
fruits = ("apple", "banana", "cherry")
```

```
print(fruits)
```

Output:

```
('apple', 'banana', 'cherry')
```

Tuple with One Item

To create a tuple with only one item, you must add a comma after the item. Otherwise, Python will not recognize it as a tuple.

Example:

```
single_item = ("apple",) # Tuple
```

```
not_a_tuple = ("apple") # Just a string
```

```
print(type(single_item)) # <class 'tuple'>
```

```
print(type(not_a_tuple)) # <class 'str'>
```

Tuple Types

1. Ordered

Tuple items are stored in a specific order.

That order will not change.

```
colors = ("red", "green", "blue")
```

```
print(colors[0])
```

Output: red

2. Unchangeable (Immutable)

Once a tuple is created, you cannot add, remove, or change its items directly.

```
numbers = (1, 2, 3)
```

```
numbers[0] = 100 # ✗ This will cause an error
```

To "change" a tuple, you can convert it to a list, make changes, and then convert it back to a tuple.

```
temp = list(numbers)
```

```
temp[0] = 100
```

```
numbers = tuple(temp)
```

```
print(numbers)
```

Output: (100, 2, 3)

3. Allows Duplicates

Tuples can have duplicate values.

```
duplicates = ("a", "b", "a", "c", "b")
```

```
print(duplicates)
```

Output:

```
('a', 'b', 'a', 'c', 'b')
```

Tuple Length

We use the len() function to find the number of items in a tuple.

Example:

```
cities = ("Srinagar", "Baramulla", "Kupwara")
```

```
print(len(cities))
```

Output: 3

Tuple with Different Data Types

A tuple can store items of different data types.

Example:

```
mixed = ("apple", 10, True, 3.14)
```

```
print(mixed)
```

Output:

```
('apple', 10, True, 3.14)
```

Summary Table:

Feature	Description	Example
Ordered	Items have a fixed order	("a", "b", "c")
Immutable	Cannot be changed after creation	my_tuple[0] = "x" ❌ Error
Allows Duplicates	Repeated values are allowed	("a", "a", "b")
One item tuple	Needs a comma: ("a",)	type(("a",)) → tuple
Length	len(tuple) returns item count	len(("x", "y")) → 2

What is a Set?

A Set is a built-in collection data type in Python that is:

Unordered: No indexing; items have no fixed position.

Unchangeable (but mutable in a way): You cannot change individual items, but you can add or remove items.

No Duplicates: Duplicate values are automatically removed.

Sets are written using curly brackets {}

Creating a Set

Syntax:

```
my_set = {item1, item2, item3}
```

Example:

```
colors = {"red", "green", "blue"}
```

```
print(colors)
```

Output (may vary in order):

```
{'blue', 'green', 'red'}
```

Set with Duplicate Items

Python automatically removes duplicates in a set.

Example:

```
nums = {1, 2, 2, 3, 4, 4}
```

```
print(nums)
```

Output:

```
{1, 2, 3, 4}
```

Creating an Empty Set

Use the set() function.

DO NOT use {} → It creates an empty dictionary, not a set.

Example:

```
empty_set = set()
```

Set Types

1 Unordered

Items have no fixed order.

You cannot access items by index like in lists or tuples.

```
fruits = {"apple", "banana", "cherry"}
```

```
# print(fruits[0]) ❌ Error
```

2 Unchangeable (Immutable items)

You cannot change individual items.

But you can add or remove items using methods like add(), remove().

3 No Duplicates Allowed

Any repeated item will be removed automatically.

Set Length

Use the len() function to find the number of items in a set.

Example:

```
cities = {"Srinagar", "Kupwara", "Baramulla"}
```

```
print(len(cities))
```

Output: 3

Adding and Removing Items

Add an Item:

```
my_set = {"a", "b"}
```

```
my_set.add("c")
```

```
print(my_set)
```

output

```
{'c', 'b', 'a'}
```

Remove an Item:

```
my_set.remove("b")
```

Set with Different Data Types

Sets can store mixed types of data.

Example:

```
mixed = {"apple", 3.14, True, 7}
```

```
print(mixed)
```

output

```
{'apple', 3.14, True, 7}
```

Summary

Feature	Description	Example
Unordered	No fixed order, no indexing	<code>{"a", "b", "c"}</code> → Order may vary
Unchangeable items	Cannot modify items directly	<code>my_set[0] = "x"</code> ❌ Error
Add/Remove allowed	Can use <code>add()</code> / <code>remove()</code> / <code>discard()</code>	<code>my_set.add("x")</code>
No Duplicates	Duplicate items removed automatically	<code>{1, 2, 2}</code> → <code>{1, 2}</code>
Empty set	Use <code>set()</code> instead of <code>{}</code>	<code>type(set())</code> → <code>set</code> , <code>type({})</code> → <code>dict</code>
Mixed types	Supports different data types	<code>{1, "apple", 3.14}</code>

UNIT 5 : Database and SQL

What is a Database?

A database is a collection of related data that is stored in an organized manner so that it can be easily accessed, managed, and updated.

A Database Management System (DBMS) is software that helps in creating, managing, and using databases.

A database is a powerful tool for handling large volumes of data efficiently. It plays an important role in education systems, banking, hospitals, companies, and more by providing secure, reliable, and fast access to data.

Examples: MySQL, Oracle, MS Access, PostgreSQL, SQLite

◆ Features of a Database:

- Stores large amounts of information
- Organizes data using tables, fields, and records
- Allows easy data search, retrieval, and updates
- Supports multiple users.

◆ Advantages of Using a Database:

- Data Organization

Stores data in a structured form using tables.

Makes data easier to read, update, and manage.

- Reduced Data Redundancy

Avoids duplication of data by using relationships between tables.

- Data Consistency and Accuracy

Ensures that data remains correct and consistent across the system.

- Data Security

Allows setting of passwords and user permissions to control access.

- Easy Data Sharing

Multiple users can access the same database at the same time without conflict.

- Faster Data Access

Powerful search and indexing features make data retrieval fast and efficient.

- Backup and Recovery

Modern DBMS supports automatic backup and restoration in case of data loss.

- Data Integrity

Ensures correctness of data using rules (constraints), like not allowing empty values for required fields.

- Data Independence

The structure of data can be changed without affecting the application programs.

- Supports Transactions

Ensures that a set of operations (like bank transfer) is completed fully or not at all.

What is a Relational Data Model?

The Relational Data Model is a way of organizing data into tables (also called relations). It is the most commonly used data model in databases and is based on the concept of relations (tables) made up of rows and columns.

Developed by E.F. Codd in 1970, the relational model is the foundation of Relational Database Management Systems (RDBMS) like:

- MySQL
- Oracle
- SQL Server
- PostgreSQL

Key Concepts of Relational Data Model:

Relation (Table):

A table with rows and columns.

Each relation has a unique name.

Tuple (Row):

A single record in a table.

Each row represents one item or object.

Attribute (Column):

A column in a table representing a property of the object.

Each attribute has a unique name.

✓ Domain:

The set of valid values an attribute can have.

Example: The domain of the "Age" column may be from 0 to 120.

✓ Primary Key:

A column (or a set of columns) that uniquely identifies each row in a table.

Example: Roll number in a Student table.

✓ Foreign Key:

A column that creates a relationship between two tables.

It refers to the primary key of another table.

Relational Data Model

◆ What is a Data Model?

A data model is a way of organizing and structuring data so it can be easily stored and accessed.

◆ What is the Relational Data Model?

The Relational Data Model is a type of data model where data is stored in the form of tables (also called relations).

Each table represents an entity (like Student, Employee, Product) and consists of rows and columns.

Rows are called records or tuples

Columns are called fields or attributes, it was introduced by E.F. Codd in 1970

◆ Example: Student Table

Roll_No	Name	Class	Marks
101	Aamir	12th	480
102	Mehak	12th	489
103	Zubair	12th	490

Each row is a student record.

Each column is an attribute like Roll_No, Name, etc.

◆ **Key Features of Relational Data Model:**

- Data is organized in tables (relations)
- Each row is unique and represents a single record
- Each column has a unique name and data type
- Relationships between tables are established using keys (Primary key, Foreign key)

◆ **Important Terms:**

Term	Description
Relation	A table in the database
Tuple	A row in the table (record)
Attribute	A column in the table (field)
Primary Key	A unique identifier for each record (e.g., Roll_No)
Foreign Key	A field that links to the primary key in another table

◆ **Advantages of Relational Data Model:**

- Simplicity – Easy to understand and use.
- Flexibility – Easy to add, update, and delete records.
- Data Integrity – Maintains accuracy through rules (constraints).
- Data Independence – Data and programs are independent of each other.
- Data Security – Permissions and roles can be set for user access.
- Supports SQL – Standard language (SQL) is used for querying.

The Relational Data Model is the most widely used model in databases today. It stores data in tables, maintains relationships using keys, and provides a powerful way to manage structured data using SQL.

1. Domain

A domain is the set of valid values that an attribute (column) can have.

Example:

In a table of students, the domain of the "Marks" attribute could be all numbers between 0 and 500.

Attribute: Marks

Domain: 0 to 500 (only numeric values)

2. Relation

A relation is simply a table that stores data in rows and columns.

Example: Student Table

Roll_No	Name	Class	Marks
101	Aamir	12th	450
102	Mehak	12th	470

Here, Student is a relation.

3. Attribute

An attribute is a column in a table. It describes a property of the entity.

Example:

In the above Student table:

Attributes are: Roll_No, Name, Class, Marks

4. Tuple

A tuple is a row in a table. It represents one record or entry.

Example:

(101, Aamir, 12th, 450) → This is a tuple.

5. Candidate Key

A candidate key is any attribute or a set of attributes that can uniquely identify each row in the table.

Example:

In the Student table, both Roll_No and Name might be unique.

So,

Roll_No → Candidate Key

Name (if unique) → Candidate Key

From candidate keys, we choose one as the primary key.

6. Primary Key

A primary key is the main unique identifier for each record in a table.
It must be unique and not null.

Example:

In the Student table, Roll_No is the primary key because each student has a different roll number.

PRIMARY KEY: Roll_No

7. Alternate Key

All candidate keys not chosen as the primary key are called alternate keys.

Example:

Candidate Keys: Roll_No, Name

Primary Key: Roll_No

So, Name becomes the Alternate Key (if it's unique)

Summary Table

Term	Description	Example
Domain	Set of valid values for an attribute	Marks: 0–500
Relation	A table	Student
Attribute	A column in a table	Name, Class, Marks
Tuple	A row in a table	(101, Aamir, 12th, 450)
Candidate Key	Attribute(s) that can uniquely identify a row	Roll_No, Name (if unique)
Primary Key	Chosen candidate key for unique ID	Roll_No
Alternate Key	Other candidate key(s) not chosen as primary	Name (if not chosen)

SQL (Structured Query Language)

Definition:

SQL is a standard programming language used to manage and manipulate relational databases. It allows users to create, retrieve, update, and delete data in a database.

Main Features of SQL:

1. Data Querying:

Allows users to fetch specific data using the SELECT statement.

2. Data Manipulation:

Data can be added (INSERT), changed (UPDATE), or removed (DELETE) from tables.

3. Data Definition:

Tables and databases can be created or modified using CREATE, ALTER, and DROP statements.

4. Data Control:

Controls access to data through GRANT and REVOKE.

5. Transaction Control:

Helps manage multiple operations using COMMIT, ROLLBACK, and SAVEPOINT.

Advantages of SQL

1. Easy to Learn and Use

SQL uses English-like syntax, which makes it easy to understand.

Example:

```
SELECT * FROM Students WHERE Class = 12;
```

This retrieves all students from class 12.

2. Efficient Data Management

SQL allows fast and efficient management of large amounts of data.

Operations like sorting, filtering, and joining multiple tables are handled easily.

3. Multiple Data Views

SQL supports views, which allow different users to see the data in different formats based on their needs.

4. High Security

SQL provides commands to control user access to data.

You can allow some users to view data, while others can modify it.

5. Portability

SQL is supported by all major database systems like MySQL, Oracle, SQL Server, PostgreSQL, etc. This makes it easy to work on different systems.

6. Standardized Language

SQL is an ANSI (American National Standards Institute) and ISO (International Organization for Standardization) standard language.

This means it is widely accepted and used globally.

7. Supports Transactions

SQL supports transactions that allow multiple operations to be performed safely.

You can commit a transaction if everything is correct or rollback if there's an error.

Conclusion:

SQL is a powerful and essential tool in the world of databases. Its simplicity, efficiency, and wide usage make it a crucial part of database management, especially useful for students and professionals alike.

Data Types in SQL

Data types define the type of data that can be stored in a table column. SQL supports several data types, but the most commonly used are:

- Number (Numeric)
- Character (Text/String)
- Date and Time

1. Number Data Types

Used to store numeric values (like roll number, marks, salary, etc.).

Data Type	Description	Example
INT	Stores whole numbers (integers)	10, 500, -23
FLOAT	Stores decimal numbers (floating point)	3.14, 99.99
DECIMAL(p,s)	Stores fixed point numbers ☐ p = Total digits allowed ☐ s = Digits after the decimal point	DECIMAL(5,2) → 123.45
SMALLINT	Stores smaller range of integers	1200

Example:

```
CREATE TABLE Students (
```

```
RollNo INT,
```

```
Marks DECIMAL(5,2));
```

2. Character Data Types

Used to store text, letters, symbols, or a combination (like names, addresses).

Data Type	Description	Example
CHAR(n)	Fixed-length string (n characters)	'Ayman', 'Yes '
VARCHAR(n)	Variable-length string (up to n characters)	'Srinagar', 'JKBOSE'
TEXT	Large amount of text (depends on DBMS)	Paragraphs

Example:

```
CREATE TABLE Students (
```

```
Name VARCHAR(50),
```

```
Gender CHAR(1)
```

```
);
```

3. Date and Time Data Types

Used to store dates and times (like date of birth, admission date).

Data Type	Description	Example
DATE	Stores date in format YYYY-MM-DD	'2025-05-21'
TIME	Stores time in format HH:MM:SS	'14:30:00'
DATETIME	Stores both date and time	'2025-05-21 14:30:00'

Example:

```
CREATE TABLE Students (
```

```
DOB DATE,
```

```
Admission Date DATETIME);
```

Summary Table:

Category	Data Types	Example
Number	INT, FLOAT, DECIMAL	100, 45.75
Character	CHAR, VARCHAR, TEXT	'Ayman', 'JKBOSE'
Date/Time	DATE, TIME, DATETIME	'2025-05-21'

1. Data Definition Language (DDL)

DDL is used to define and manage the structure of a database – like creating or deleting tables.

♦ Common DDL Commands:

Command	What it does	Example
CREATE	Creates a new table or database	CREATE TABLE Students (...);
ALTER	Changes structure of a table	ALTER TABLE Students ADD Age INT;
DROP	Deletes a table or database	DROP TABLE Students;

DDL commands change the structure, not the data.

2. Data Manipulation Language (DML)

DML is used to insert, update, or delete the data inside the tables.

♦ Common DML Commands:

Command	What it does	Example
INSERT	Adds new data into a table	INSERT INTO Students VALUES (...);
UPDATE	Changes existing data	UPDATE Students SET Age = 18 WHERE RollNo = 1;
DELETE	Removes data from a table	DELETE FROM Students WHERE RollNo = 1;
SELECT	Fetches data from the table	SELECT * FROM Students;

DML is used for working with the data stored in tables.

Difference Between DDL and DML

DDL (Structure)	DML (Data)
Defines structure of tables	Works with the actual data
Uses CREATE, ALTER, DROP	Uses INSERT, UPDATE, DELETE, SELECT
Affects structure only	Affects the content

1. DDL Commands (Data Definition Language)

Used to define or change the structure of a database (tables, columns, etc.)

◆ Common DDL Commands:

CREATE : To Create New Table

CREATE TABLE Students (

RollNo INT,

Name VARCHAR(50),

Age INT

);

ALTER To change the structure of a table (like adding a column)

ALTER TABLE Students ADD Gender CHAR(1);

DROP To delete a table

DROP TABLE Students;

TRUNCATE To delete all data from a table but keep the table

TRUNCATE TABLE Students;

2. DML Commands (Data Manipulation Language)

Used to work with data inside tables.

◆ Common DML Commands:

INSERT : To add new data into a table

INSERT INTO Students (RollNo, Name, Age, Gender)

VALUES (1, 'Ayman', 17, 'M');

UPDATE : To change existing data in a table |

UPDATE Students SET Age = 18 WHERE RollNo = 1;

DELETE : To remove data from a table |

DELETE FROM Students WHERE RollNo = 1;

SELECT : To get data from a table

SELECT * FROM Students;

Summary Table

Type	Commands	Purpose
DDL	CREATE, ALTER, DROP, TRUNCATE	Define or change structure
DML	INSERT, UPDATE, DELETE, SELECT	Add or manage data

SQL Functions

SQL functions are used to perform calculations on data in a table.

◆ **1. SUM()** – Total of values

Use: Adds all the values in a column.

Example:

SELECT SUM(Marks) FROM Students;

This will give the total marks of all students.

◆ **2. AVG()** – Average value

Use: Finds the average (mean) of values.

Example:

SELECT AVG(Marks) FROM Students;

This will give the average marks of all students.

◆ **3. COUNT()** – Count rows

Use: Counts the number of rows (records).

Example:

SELECT COUNT(*) FROM Students;

→ This will count total number of students.

◆ **4. MIN()** – Minimum value

Use: Finds the smallest value in a column.

Example:

SELECT MIN(Marks) FROM Students;

→ This will show the lowest marks in the class.

◆ **5. MAX()** – Maximum value

Use: Finds the largest value in a column.

Example:

SELECT MAX(Marks) FROM Students;

→ This will show the highest marks in the class.

■ Sample Table: Students

RollNo	Name	Marks
1	Ayman	80
2	Zoya	90
3	Irfan	70

SUM(Marks) → 240

AVG(Marks) → 80

COUNT(*) → 3

MIN(Marks) → 70

MAX(Marks) → 90

Unit 6: Boolean Logic

Introduction to Boolean Logic

Boolean Logic is a fundamental concept in computer science and digital electronics. It deals with binary values—True and False (or 1 and 0). It is used in decision making, conditional statements, logical circuits, and program control.

Boolean logic is named after George Boole, a mathematician who laid the foundations of mathematical logic in the 19th century.

Boolean Values

In programming (including Python), Boolean values are:

True → Logic 1

False → Logic 0

Examples in Python:

```
x = 10
```

```
y = 5
```

```
print(x > y)
```

```
# Output: True
```

```
print(x == y)
```

```
# Output: False
```

Boolean Operators

Boolean operators are used to combine or modify Boolean expressions. There are three basic Boolean operators:

AND Operator (and)

Returns True if both conditions are true.

Returns False if any one or both are false.

Syntax in Python:

A and B

Truth Table:

A	B	A AND B
True	True	True
True	False	False
False	True	False
False	False	False

Example:

```
x = 15
```

```
print(x > 10 and x < 20)
```

Output : True

✔ OR Operator (or)

Returns True if at least one condition is true.

Returns False only if both conditions are false.

Syntax:

```
python
```

```
A or B
```

Truth Table:

A	B	A OR B
True	True	True
True	False	True
False	True	True
False	False	False

Example:

```
x = 25
```

```
print(x < 10 or x > 20)
```

Output: True

✅ NOT Operator (not)

Returns the opposite of the Boolean value.

not True → False

not False → True

Syntax:

python

not A

Example:

x = 7

print(not(x > 10))

Output: True

(because x > 10 is False, and not False = True)

◆ 4. Use of Boolean Logic in Programming

In Python and other languages, Boolean logic is used in:

if/else statements

loop control

error handling

decision-based programs

Example in Python:

age = 18

has_voter_id = True

if age >= 18 and has_voter_id:

 print("Eligible to vote")

else:

 print("Not eligible")

◆ 6. Boolean Expressions in Conditions

Any expression that evaluates to True or False can be part of a Boolean logic expression.

```
a = 10
```

```
b = 20
```

```
print((a < b) and (b > 15))
```

Output: True

```
print((a > b) or (b == 20))
```

Output: True

Summary Table

Operator	Description	Symbol in Python
AND	True if both conditions are true	and
OR	True if at least one is true	or
NOT	Reverses the logical state	not

Truth Table:

A truth table is a table that shows all possible inputs and their corresponding outputs for a Boolean operator like AND, OR, or NOT.

Truth Table for AND (and)

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Truth Table for OR (or)

A	B	A or B
True	True	True
True	False	True

False	True	True
False	False	False

Truth Table for NOT (not)

A	not A
True	False
False	True

In Short: AND → True only when both are True

OR → True if any one is True

NOT → Opposite value

What is Boolean Algebra?

Boolean Algebra is a mathematical system used to work with binary values (True/False or 1/0). It is used in logic circuits, programming, and digital electronics.

Basic Boolean Values:

0 means False

1 means True

Important Boolean Laws (with Truth Table Proofs)

1. Commutative Law

This law states that the order of the variables does not affect the result.

a) For OR:

$$A + B = B + A$$

A	B	A + B	B + A
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Both columns are equal → OR is commutative

b) For AND:

$$A \cdot B = B \cdot A$$

A	B	$A \cdot B$	$B \cdot A$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Both columns are equal $\rightarrow$ AND is commutative

2. Associative Law

This law says that grouping of variables does not affect the result.

a) For OR:

$$A + (B + C) = (A + B) + C$$

A	B	C	$B+C$	$A+(B+C)$	$A+B$	$(A+B)+C$
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

LHS = RHS $\rightarrow$ OR is associative

b) For AND:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

A	B	C	$B \cdot C$	$A \cdot (B \cdot C)$	$A \cdot B$	$(A \cdot B) \cdot C$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	1	0
1	1	1	1	1	1	1

LHS = RHS $\rightarrow$ AND is associative

✔ 3. Distributive Law

This law connects AND and OR operations together.

a) AND over OR:

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$$

A	B	C	$B+C$	$A \cdot (B+C)$	$A \cdot B$	$A \cdot C$	$A \cdot B + A \cdot C$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

LHS = RHS $\rightarrow$ Distributive law holds

✓ 4. Principle of Duality

Definition:

Every Boolean expression remains valid if we interchange AND and OR and replace 0 with 1 and 1 with 0.

Example:

Original: $A + 0 = A$

Dual: $A \cdot 1 = A$ ✓ Both are valid.

Original: $A + A = A$

Dual: $A \cdot A = A$ ✓ Valid

Original: $A + A' = 1$

Dual: $A \cdot A' = 0$ ✓ Valid

Truth tables prove both the original and its dual are logically correct.

Summary Table of Laws:

Law Name	Expression	Dual Form
Commutative (OR)	$A + B = B + A$	—
Commutative (AND)	$A \cdot B = B \cdot A$	—
Associative (OR)	$A + (B + C) = (A + B) + C$	—
Associative (AND)	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	—
Distributive	$A \cdot (B + C) = AB + AC$	$A + (BC) = (A + B)(A + C)$
Duality Principle	Replace + with ; 0 with 1	and vice versa

What are DeMorgan's Laws?

DeMorgan's Laws are two important rules in Boolean algebra that show how negation (NOT) interacts with AND and OR operations.

They are used in simplifying Boolean expressions, designing logic circuits, and programming conditional statements.

DeMorgan's Laws – Statements

♦ **First Law:**

The complement of (A AND B) is equal to (NOT A) OR (NOT B)

$$\neg(A \cdot B) = \neg A + \neg B$$

In Python-style:

not (A and B) == (not A) or (not B)

♦ **Second Law:**

The complement of (A OR B) is equal to (NOT A) AND (NOT B)

$$\neg(A + B) = \neg A \cdot \neg B$$

In Python-style:

not (A or B) == (not A) and (not B)

Truth Table Proofs

✓ First Law: $\neg(A \cdot B) = \neg A + \neg B$

A	B	A · B	$\neg(A \cdot B)$	$\neg A$	$\neg B$	$\neg A + \neg B$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

LHS and RHS columns match → First DeMorgan's Law is proven

✓ Second Law: $\neg(A + B) = \neg A \cdot \neg B$

A	B	A+B	$\neg(A+B)$	$\neg A$	$\neg B$	$\neg A \cdot \neg B$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

LHS and RHS columns match $\rightarrow$ Second DeMorgan's Law is proven

Unit 7: Networking and Cyber Security

Computer networking is the practice of connecting two or more computers or devices together so they can share information, resources (like printers or files), and communicate with each other.

Advantages of Computer Networking

Resource Sharing

Devices like printers, scanners, and storage can be shared across a network, saving cost.

File Sharing

Files and data can be easily shared between connected devices, increasing efficiency.

Communication

Networking allows for quick communication through emails, instant messaging, video conferencing, etc.

Centralized Data Management

Data can be stored and managed from a central server, improving security and access control.

Cost Efficiency

Shared resources and centralized management reduce the need for duplicate devices and storage.

Remote Access

Users can access network resources and data from remote locations.

Scalability

New devices can be easily added to the network as needed.

Improved Data Security

Centralized systems can implement better security measures like firewalls, encryption, and access controls.

Types of Computer Networks

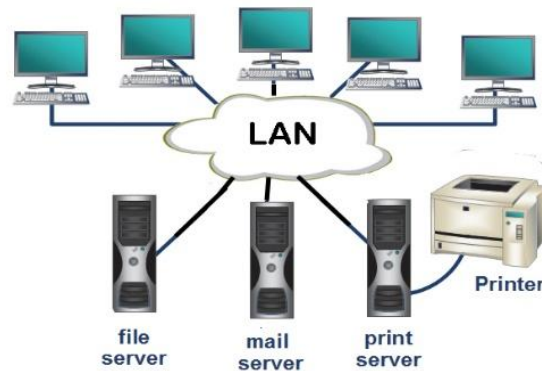
One way of classifying the different types of computer network is by their scope or scale. Common example of different types of network is as.

-> **LAN (Local Area Network.)**

- > WAN (Wide Area Network.)
- > MAN (Metropolitan Area Network.)
- > PAN (Personal Area Network.)

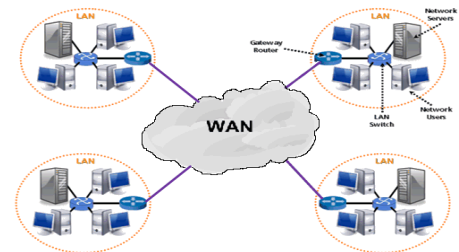
LAN (Local Area Network).

A LAN connects network devices over a relatively short distance. For example a network present in school, home, office building is an example of local area network.



WAN (Wide Area Network)

As the name of implies, a WAN ranges over a large distance .For example internet is the largest WAN, distributed across the entire earth. A WAN is collection of many LAN'S.A network connecting device router connects LAN'S to WAN'S.



MAN (Metropolitan Area network) .

This kind of network is not mostly used but it has its own importance for some government bodies and organizations on large scale MAN network falls in middle of LAN and WAN. It covers large distance than LAN but smaller than WAN such as city.

PAN(Personal Area Network).

This type of network is also called Pico network. This type of network is used for communication among the personal devices i.e. interpersonal communication.



For example a wireless network formed by using Bluetooth/infrared technology between laptop and other devices like Bluetooth mouse, printer etc is an example of wireless PAN.



Network topology

Network topology refers to the arrangement or layout of different elements (like computers, links, and devices) in a computer network. It describes how the network components are physically or logically connected to each other.

There are two main types of network topology:

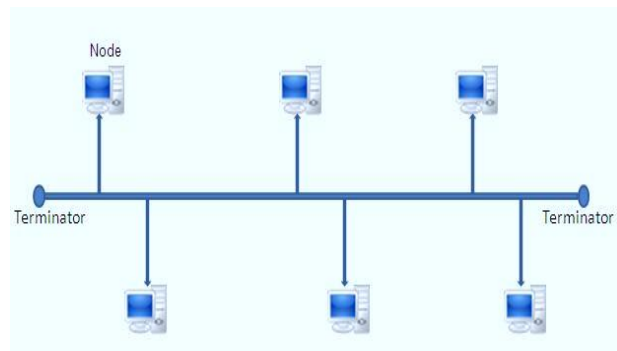
Physical topology – the actual physical layout of the devices and cables.

Logical topology – the way data moves through the network, regardless of its physical design.

Common Types of Network Topologies:

Bus topology

Bus topology is one of the simplest types of network topologies. In a bus topology, all devices (computers, printers, etc.) are connected to a single central cable, called the bus or backbone.



Key Features:

Single communication line: Data travels in both directions along the bus.

Terminators are placed at both ends of the cable to prevent signal bounce.

Devices listen for data addressed to them and ignore the rest.

Advantages:

Easy to set up.

Requires less cable compared to other topologies.

Cost-effective for small networks.

Disadvantages:

Performance degrades with more devices or heavy traffic.

Difficult to troubleshoot.

If the main cable (bus) fails, the whole network goes down.

Star topology

In a star topology, all devices (computers, printers, etc.) are individually connected to a central device, such as a hub, switch, or router.

This central device acts as a controller and manages the flow of data.



Key Features:

Each device has a dedicated cable connecting it to the central hub.

Data is sent from one device to another via the central hub/switch.

Advantages:

Easy to install and manage.

Failure of one device doesn't affect the others.

Easy to detect and fix faults (just check the connection to the hub).

Disadvantages:

If the central hub fails, the whole network goes down.

Requires more cable than bus topology.

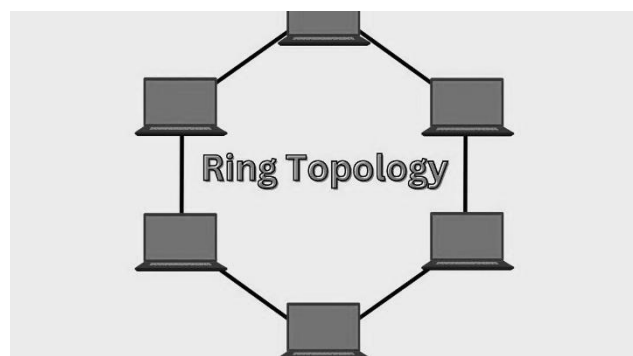
Can be more expensive due to the cost of the hub and cables.

Ring topology

Ring topology is a type of network setup where each device is connected to exactly two other devices, forming a circular path for data to travel.

Key Features:

Devices are connected in a closed loop.



Data flows through each device until it reaches the intended recipient.

Often used in older networks or in token ring systems.

Advantages:

Data flows in a predictable direction.

Easier to manage than bus when the network is large.

Can perform better than bus topology under heavy load.

Disadvantages:

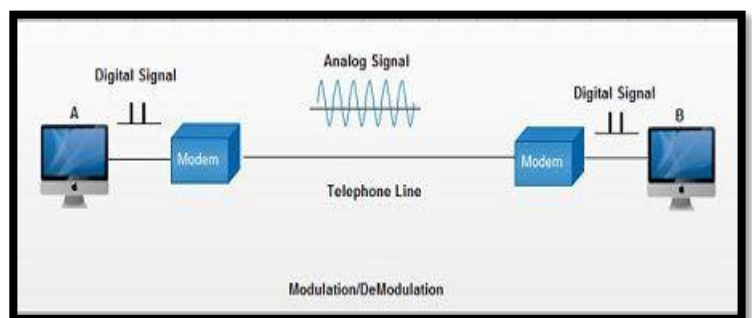
Failure of one device or cable can break the whole network (unless it's a dual ring).

Adding or removing devices can disrupt the network.

Troubleshooting is harder compared to star topology.

MODEM

A modem (short for Modulator-Demodulator) is a network device that connects your computer or local network to the internet by converting digital signals into analog signals and vice versa.



Working:

Modulation: Converts digital data from your computer into analog signals to send over telephone or cable lines.

Demodulation: Converts incoming analog signals from the internet back into digital data that your computer can understand.

Cyber Safety

Cyber Safety refers to safe and responsible behaviour while using digital devices and the internet. It's mainly focused on protecting individuals, especially children and young users, from online threats and harmful behaviour.

Key Aspects of Cyber Safety:

- Avoiding cyberbullying and reporting it.
- Being cautious while sharing personal information online.
- Not clicking on suspicious links or downloading unknown attachments.

- Using strong and unique passwords.
- Being aware of scams like phishing and fake profiles.

Cyber Security

Cyber Security is more technical and refers to protecting computers, networks, programs, and data from unauthorized access, attacks, or damage.

Key Aspects of Cyber Security:

- Installing and updating antivirus software.
- Using firewalls and encryption.
- Keeping systems and apps updated (patch management).
- Securing Wi-Fi networks.
- Preventing hacking, ransomware, spyware, etc.

Example:

Cyber Safety: A student avoids talking to strangers online and reports a suspicious message to a teacher.

Cyber Security: The school's IT department installs security software to prevent hackers from accessing school data.

What is Cyberbullying?

Cyberbullying is when someone uses digital devices, such as mobile phones, computers, or tablets, to intentionally hurt, harass, threaten, or embarrass another person.

It can happen on:

- Social media (like Facebook, Instagram, Snapchat)
- Messaging apps (like WhatsApp, Messenger)
- Online games
- Emails and text messages.

Examples of Cyberbullying:

- Sending mean or threatening messages
- Spreading rumours online
- Posting embarrassing photos or videos

- Creating fake profiles to mock someone
- Excluding someone from an online group

General Preventive Measures for Cyberbullying

1. Protect Personal Information

Never share passwords, home address, phone number, or school/work details publicly.

Use strong, unique passwords and change them regularly.

2. Think Before You Post

Avoid sharing anything online that can hurt you or others.

Ask yourself: "Would I say this in person?"

3. Use Privacy Settings

Set your profiles, posts, and chats to "private."

Control who can see your posts or contact you.

4. Don't Engage with Bullies

Do not reply to hateful or rude comments.

Responding can make the situation worse.

5. Report and Block

Report abusive accounts and block the bully on the platform.

Use platform safety tools to filter offensive content.

6. Keep Evidence

Take screenshots or save messages/posts as proof in case action needs to be taken.

7. Be a Responsible Digital Citizen

Use technology to spread kindness and awareness.

Respect others online, just like you would in real life.

8. Speak Up

If you or someone else is being bullied, tell a trusted adult, teacher, or report it to the platform.

Encourage a culture of reporting and support.

9. Create Awareness Together

Schools, families, and communities should work together to educate people about the risks and solutions.

What is Internet Safety?

Internet safety refers to practicing safe and responsible behaviour while using the internet to protect yourself and your personal information from harm. Some best practices are as under:

- Use strong passwords – Mix letters, numbers, symbols.
- Don't share personal info – Keep details private.
- Avoid suspicious links – Beware of scams and phishing.
- Check for "https" – Use secure websites.
- Update software – Stay protected.
- Think before posting – Internet remembers everything.
- Use antivirus & firewall – Block harmful threats.
- Report problems – Tell an adult or report online.

What is internet Ethics

Internet Ethics refers to the set of moral principles and rules that guide how people should behave and interact online as given below.

- Be Respectful – Treat others kindly online.
- Avoid Cyberbullying – Never hurt or insult anyone.
- Don't Plagiarize – Always give credit for others' work.
- Use the Internet Legally – Don't hack, steal, or pirate content.
- Protect Privacy – Respect others' personal info.
- Be Honest – Don't spread lies or fake news.
- Use Technology for Good – Help, educate, and inspire others.

Safe Social Networking

Safe Social Networking means using social media in a way that protects your privacy, security, and well-being as given below:

- Protect your personal information
- Use strong and private passwords
- Don't accept strangers

- Think before you post
- Avoid sharing live location
- Report/block anything suspicious.

Safe Email Practices

Safe Email Practices are the rules to follow to keep your email secure and avoid scams or viruses as mentioned below.

- Don't open unknown emails or links
- Avoid downloading unknown attachments
- Use strong passwords
- Log out after using email on shared devices
- Don't share personal info over email
- Enable spam filters and two-factor authentication (2FA)
- Check sender's email address carefully

Do's and Don'ts for Cyber Safety:

Do's:

- Use strong, unique passwords
- Keep software and antivirus updated
- Think before you click on links or ads
- Use privacy settings on social media
- Report cyberbullying or suspicious activity
- Log out from public/shared devices
- Backup important data regularly

Don'ts

- Don't share personal information online
- Don't use weak or common passwords
- Don't click on suspicious links or pop-ups
- Don't open unknown email attachments
- Don't talk to or accept friend requests from strangers
- Don't download files from untrusted websites
- Don't post private details on social media