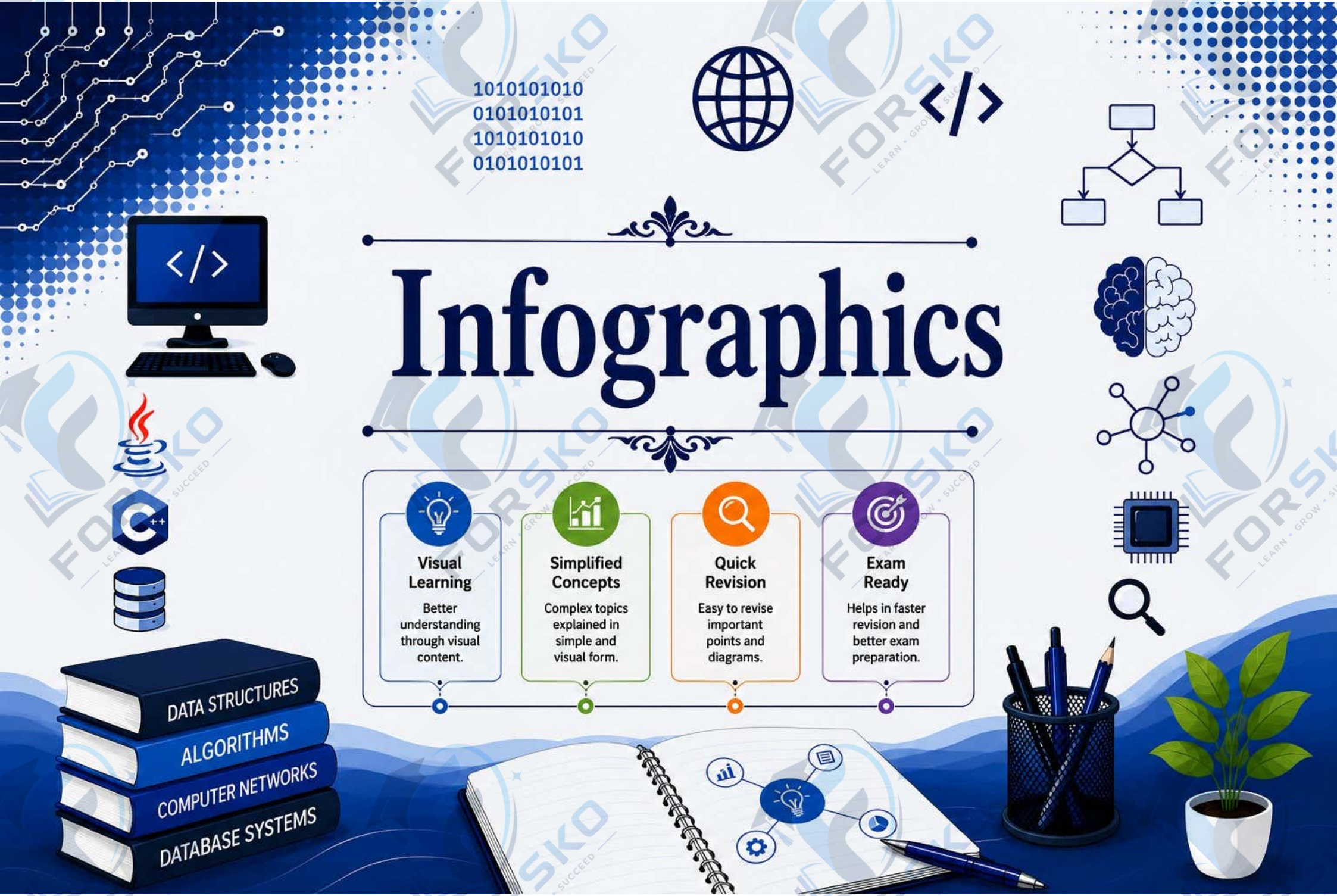

1010101010
0101010101
1010101010
0101010101
Infographics
Visual Learning
Better understanding through visual content.
Simplified Concepts
Complex topics explained in simple and visual form.
Quick Revision
Easy to revise important points and diagrams.
Exam Ready
Helps in faster revision and better exam preparation.
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS


1010101010
0101010101
1010101010
0101010101
Unit-I
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

OOP PARADIGM: BASIC CONCEPTS

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "Objects", which can contain data in the form of fields (attributes) and code in the form of methods (functions).

THE FOUR PILLARS OF OOP

1 ENCAPSULATION

Binding data (attributes) and methods into a single unit (class) and restricting access to some of the object's components.

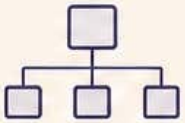
It helps in data hiding and protects data from unwanted access.

2 ABSTRACTION

Hiding complex implementation details and showing only the essential features to the user.

It helps in reducing complexity and focuses on what an object does, not how it does it.

3 INHERITANCE



Creating new classes from existing classes. The new class (derived class) inherits attributes and methods from the existing class (base class).

It promotes code reusability.

4 POLYMORPHISM

The ability of an object to take many forms. The same method (or function) can behave differently for different objects.

It provides flexibility and extensibility.

CLASS vs OBJECT

CLASS (Blueprint)

A template or blueprint for creating objects.

Car
- brand - color - speed
+ start() + stop()

Defines properties and behaviors.

Objects are created from classes.

OBJECT (Instance)

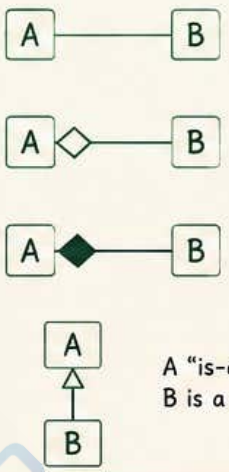
An instance of a class. It is a real-world entity.

car1

brand = "Toyota"
color = "Red"
speed = 60

RELATIONSHIPS (Types of Relationships)

- **Association** — A general relationship between two classes.
- **Aggregation** — A "has-a" relationship (weak ownership).
- **Composition** — A "has-a" relationship (strong ownership).
- **Inheritance** — A "is-a" relationship. B is a specialized form of A.



BENEFITS OF OOP

- **Reusability** – Code can be reused using inheritance.
- **Maintainability** – Easy to maintain and modify existing code.
- **Modularity** – Program is divided into small, independent objects.
- **Scalability** – Easy to extend and scale applications.
- **Real-world modeling** – Maps real-world entities and relationships.

★ In OOP, we don't just write code; we design and build real-world models using classes and objects.

OOP PARADIGM: BASIC CONCEPTS

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "Objects", which can contain data in the form of fields (attributes) and code in the form of methods (functions).

THE FOUR PILLARS OF OOP

1 ENCAPSULATION

Binding data (attributes) and methods into a single unit (class) and restricting access to some of the object's components.

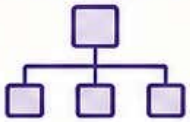
It helps in data hiding and protects data from unwanted access.

2 ABSTRACTION

Hiding complex implementation details and showing only the essential features to the user.

It helps in reducing complexity and focuses on what an object does, not how it does it.

3 INHERITANCE



Creating new classes from existing classes. The new class (derived class) inherits attributes and methods from the existing class (base class).

It promotes code reusability.

4 POLYMORPHISM

The ability of an object to take many forms. The same method (or function) can behave differently for different objects.

It provides flexibility and extensibility.

FEATURES OF OOP

Class and Object
A class is a blueprint for creating objects. An object is an instance of a class.

Data Abstraction
Shows only essential information and hides unnecessary implementation details.

Data Encapsulation
Bundles data and methods together and restricts direct access to some components.

Inheritance
Allows creating new classes from existing classes to reuse and extend functionality.

Polymorphism
Allows objects of different classes to be treated as objects of a common base class.

Dynamic Binding
The method call is resolved at runtime, not at compile time.

ADVANTAGES OF OOP

Modularity
Program is divided into smaller modules (classes), making it easy to develop and maintain.

Code Reusability
Inheritance and polymorphism allow reuse of existing code, reducing development time.

Maintainability
Easy to modify and update existing code without affecting other parts.

Security
Data hiding and encapsulation protect data from unintended access and misuse.

Scalability
Easy to extend and scale the application as requirements grow.

Real-world Modeling
OOP represents real-world entities and relationships more naturally and effectively.

EXAMPLE

Class: Car

Attributes (Data)
- brand
- color
- speed

Methods (Functions)
- start()
- stop()
- display()

Object: car1

brand = "Toyota"
color = "Red"
speed = 60

In OOP, we don't just write code; we design and build real-world models using classes and objects.

OOP PARADIGM: BASIC CONCEPTS

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "Objects", which can contain data in the form of fields (attributes) and code in the form of methods (functions).

THE FOUR PILLARS OF OOP

1 ENCAPSULATION

Binding data (attributes) and methods into a single unit (class) and restricting access to some of the object's components.

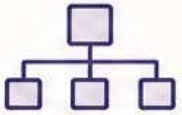
It helps in data hiding and protects data from unwanted access.

2 ABSTRACTION

Hiding complex implementation details and showing only the essential features to the user.

It helps in reducing complexity and focuses on what an object does, not how it does it.

3 INHERITANCE



Creating new classes from existing classes. The new class (derived class) inherits attributes and methods from the existing class (base class).

It promotes code reusability.

4 POLYMORPHISM

The ability of an object to take many forms. The same method (or function) can behave differently for different objects.

It provides flexibility and extensibility.

FEATURES OF OOP

Class and Object
A class is a blueprint for creating objects. An object is an instance of a class.

Data Abstraction
Shows only essential information and hides unnecessary implementation details.

Data Encapsulation
Bundles data and methods together and restricts direct access to some components.

Inheritance
Allows creating new classes from existing classes to reuse and extend functionality.

Polymorphism
Allows objects of different classes to be treated as objects of a common base class.

Dynamic Binding
The method call is resolved at runtime, not at compile time.

ADVANTAGES OF OOP

Modularity
Program is divided into smaller modules (classes), making it easy to develop and maintain.

Code Reusability
Inheritance and polymorphism allow reuse of existing code, reducing development time.

Maintainability
Easy to modify and update existing code without affecting other parts.

Security
Data hiding and encapsulation protect data from unintended access and misuse.

Scalability
Easy to extend and scale the application as requirements grow.

Real-world Modeling
OOP represents real-world entities and relationships more naturally and effectively.

APPLICATIONS OF OOP

Software Development
Desktop applications, web applications, IDE, compilers

Mobile Applications
Android, iOS apps, mobile games

Game Development
2D/3D games, game engines

Banking & Finance
Online banking, transaction systems, ATM systems

E-commerce
Online stores, payment gateways, shopping carts

Healthcare
Hospital management systems, patient records

★ In OOP, we don't just write code; we design and build real-world models using classes and objects.

STRUCTURED PROGRAMMING Vs OOP (OBJECT-ORIENTED PROGRAMMING)

Both are programming paradigms used to develop software, but they differ in approach, focus and design.

ASPECT	STRUCTURED PROGRAMMING	OOP (OBJECT-ORIENTED PROGRAMMING)
Basic Unit	Function / Procedure	Class and Object
Focus	Focuses on solving a problem by dividing it into functions.	Focuses on modeling real-world entities as objects.
Approach	Top-down approach.	Bottom-up approach.
Data and Code	Data is separate from code. Functions operate on data.	Data (attributes) and code (methods) are bundled together in a class.
Data Security	Less secure (data is global and can be accessed by any function).	More secure (data hiding using encapsulation and access specifiers).
Reusability	Limited reusability. Functions can be reused but not data easily.	High reusability through inheritance, polymorphism and encapsulation.
Maintenance	Difficult to maintain in large programs.	Easier to maintain and extend.
Scalability	Not suitable for large and complex systems.	Highly scalable for large and complex systems.
Examples (Languages)	C, Basic, Pascal	C++, Java, Python, C#, PHP, etc.
Real-world Modeling	Difficult to represent real-world entities.	Easy to model real-world entities and relationships.

QUICK EXAMPLE

STRUCTURED APPROACH

To store and print student details

Input student data
↓
Process data in functions
↓
Display student data

OOP APPROACH

Model student as an object

Class: Student

Attributes (Data)
- name
- rollNo
- marks

Methods (Functions)
- input()
- display()

Create object of Student class and call its methods.

FEATURES

STRUCTURED PROGRAMMING

- ✓ Uses functions and control structures (if, for, while).
- ✓ Program is divided into smaller functions.
- ✓ Data is global and accessible.
- ✓ Easy to understand for small programs.
- ✓ Follows step-by-step problem solving.

OOP

- ✓ Uses classes and objects.
- ✓ Supports concepts like encapsulation, inheritance, polymorphism, abstraction.
- ✓ Promotes data hiding and security.
- ✓ Code is modular, reusable and extensible.
- ✓ Closer to real-world modeling.

ADVANTAGES

STRUCTURED PROGRAMMING

- ★ Simple and easy to learn.
- ★ Efficient for small and straightforward problems.
- ★ Less memory requirement.
- ★ Fast execution.
- ★ Good for system programming.

OOP

- ★ Modularity and code reusability.
- ★ Easy maintenance and updation.
- ★ Data security through encapsulation.
- ★ Scalable for large projects.
- ★ Real-world problem modeling.

SUMMARY Structured Programming is function-centric, while OOP is object-centric. OOP provides more flexibility, reusability and scalability, making it ideal for modern software development.

KEY TAKEAWAY

- Use Structured Programming for small and simple problems.
- Use OOP for large, complex and real-world applications.

TRENDING OOP LANGUAGES </>

Object-Oriented Programming (OOP) is a powerful paradigm used to build modular, reusable and scalable applications.
Here are some of the most popular and in-demand OOP languages in 2024.

1 Python

A versatile, high-level language known for its simplicity and readability.

Key Features

- Easy to learn
- Clean and readable syntax
- Strong standard library
- Supports OOP fully
- Wide range of applications

2 Java

A robust, platform-independent language widely used in enterprise applications.

Key Features

- Write once, run anywhere
- Strong memory management
- Rich API and frameworks
- Fully object-oriented
- Excellent for large systems

3 C++

A powerful, high-performance language used in system software and games.

Key Features

- High performance
- Low-level memory access
- OOP + Generic programming
- Extensive libraries
- Used in real-time systems

4 C#

A modern, type-safe language developed by Microsoft for the .NET ecosystem.

Key Features

- Integrated with .NET
- Simple and modern syntax
- Strong tooling (Visual Studio)
- Supports OOP completely
- Great for enterprise apps and games

5 JavaScript (ES6+)

Primarily a scripting language now widely used for OOP in web and backend.

Key Features

- ES6+ class syntax
- Prototype-based OOP
- Huge ecosystem (npm)
- Used in frontend & backend
- High demand in web dev

6 Kotlin

A modern, concise language by JetBrains, fully interoperable with Java.

Key Features

- Concise and expressive
- Null safety
- Fully object-oriented
- Interoperable with Java
- Great for Android development

OOP PRINCIPLES SUPPORTED

Encapsulation	Bundling data and methods together and controlling access.
Inheritance	Creating new classes from existing classes.
Polymorphism	Same interface, different implementations.
Abstraction	Showing only essential features and hiding the implementation.
Class & Object	Blueprint (class) and real-world entity (object).

HOW THESE LANGUAGES COMPARE

Language	Performance	Ease of Learning	Industry Demand	Community Support	Versatility
Python	★★★★☆	★★★★★	★★★★★	★★★★★	★★★★★
Java	★★★☆☆	★★★☆☆	★★★★★	★★★★★	★★★★☆
C++	★★★★★	★★☆☆☆	★★★★☆	★★★★☆	★★★★☆
C#	★★★★☆	★★★★☆	★★★★☆	★★★★☆	★★★★☆
JavaScript (ES6+)	★★★☆☆	★★★★☆	★★★★★	★★★★★	★★★★★
Kotlin	★★★☆☆	★★★★☆	★★★★☆	★★★★☆	★★★★☆

★★★★★ Excellent ★★★★☆ Very Good ★★★☆☆ Good ★★☆☆☆ Average ★☆☆☆☆ Low

WHY THESE LANGUAGES ARE TRENDING

- Widely used in modern applications, from web and mobile to AI and cloud.
- High demand in job market and strong industry adoption.
- Constantly evolving with new features and Improvements.
- Large communities and rich ecosystems of libraries and frameworks.
- Suitable for scalable, maintainable and real-world solutions.

QUICK TIP

Choose the language based on your goal:
Web Development → JavaScript / Python
Enterprise Apps → Java / C#
Mobile Apps → Kotlin (Android)
System / Game Development → C++

OOP helps you write better code by making it organized, reusable, and easy to maintain.

Master OOP today, build the solutions of tomorrow!

OOP CONCEPTS – DATA ABSTRACTION AND ENCAPSULATION

Data Abstraction and Encapsulation are two fundamental OOP concepts that help in building secure, flexible and maintainable applications.

1 DATA ABSTRACTION

Data Abstraction is the process of hiding the internal implementation details and showing only the essential features of an object to the user. It helps to reduce complexity and focus on what an object does, not how it does it.

KEY POINTS

- Hides unnecessary details.
- Shows only essential features (what an object does).
- Achieved using abstract classes and interfaces.
- Improves security and reduces impact of changes.

REAL-LIFE ANALOGY

EXAMPLE (Python)

```python
from abc import ABC, abstractmethod
class Shape(ABC):                    # Abstract Class
    @abstractmethod
    def area(self):
        pass

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

c = Circle(5)
print(c.area())    # User sees only 'area' method
```

What the user sees:

```
c = Circle(5)
c.area()
```

↓

78.5

What is hidden:
The internal implementation of 'area' and calculations are hidden from the user.

2 ENCAPSULATION

Encapasulation is the process of binding data (attributes) and methods (functions) together in a single unit (class) and restricting direct access to some of the object's components. It protects data from unintended access or modification.

KEY POINTS

- Binds data and methods together.
- Restricts direct access to data.
- Achieved using access modifiers (private, protected, public).
- Improves data security and maintains integrity.

REAL-LIFE ANALOGY

You can use the ATM interface to:
- Insert card
- Enter PIN
- Withdraw money

You cannot access the internal mechanism:
- How it verifies PIN
- How it dispenses cash
- How it processes transaction

EXAMPLE (Python)

```python
class BankAccount:
    def __init__(self, owner, balance):
        self.__balance = balance    # private attribute
        self.owner = owner          # public attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):          # public method
        return self.__balance

a = BankAccount("Ali", 1000)
a.deposit(500)
print(a.get_balance())    # 1500
# print(a.__balance)      # Error! cannot access directly
```

Access Levels

Modifier	Meaning	Accessible From
public	No restriction	Anywhere
_protected	Protected (convention)	Within class & subclasses
__private	Private (restricted)	Within class only

Benefits

- Protects data from unauthorized access
- Controls how data is modified
- Improves code reliability and maintainability

DIFFERENCE AT A GLANCE

Aspect	Data Abstraction	Encapsulation
Focus	Hiding unnecessary details and showing only essential features.	Binding data and methods together and restricting access to data.
Purpose	Reduces complexity.	Protects data and maintains integrity.
Achieved By	Abstract classes, Interfaces.	Access modifiers (private, protected, public).
User Sees	What an object does.	What data can be accessed and how.
Example	Interface of a car (drive, brake).	ATM machine (use interface, internals hidden).

HOW THEY WORK TOGETHER

Data Abstraction (Hides Complexity) + Encapssulation (Protects Data) = Secure, Flexible & Maintainable Applications

KEY TAKEAWAY

Use Data Abstraction to focus on essential features for the user.
Use Encapsulation to protect data and control access.
Together, they make your OOP programs robust, secure and easy to maintain.

OOP CONCEPTS – CLASSES AND OBJECTS, DEFINING A CLASS

In Object-Oriented Programming, a Class is a blueprint for creating objects (instances). An Object is a real-world entity that has state (data) and behavior (functions).

1. CLASSES AND OBJECTS

Class (Blueprint)

- A class is a template or blueprint that defines the properties (data) and behaviors (methods) that objects of its type will have.
- It does not take memory until an object is created.

Class: Car

Properties (Data):
brand, color, model, speed

Methods (Behavior):
start(), stop(), drive()

Creates →

Object (Instance)

- An object is a real-world entity created from a class.
- It has its own identity, state (data) and behavior.
- Memory is allocated when an object is created.

Objects of Car Class

car1	car2
brand = Toyota	brand = Honda
color = Blue	color = Red
model = Corolla	model = City

Real-Life Analogy

A Class is like a blueprint of a house.
Objects are the actual houses built using that blueprint.
All houses (objects) may have same structure, but different owners, colors, etc.

Key Points

- ✓ Class is a blueprint; Object is an instance.
- ✓ Multiple objects can be created from a single class.
- ✓ Objects have unique identity and state.
- ✓ Class defines what objects will have; Objects carry the actual values.

Example (Python)

```
# Class definition
class Car:
    pass   # Empty class for now

# Creating objects of Car class
car1 = Car()
car2 = Car()

print(type(car1))   # <class '__main__.Car'>
```

Summary

Class → Blueprint / Template
Object → Real-world entity (instance of a class)

2. DEFINING A CLASS

A class is defined using the 'class' keyword.
It can contain attributes (variables) and methods (functions).

Syntax

```
class ClassName:
    # attributes (variables)
    def method1(self, parameters):
        # method body
        pass
    def method2(self):
        # method body
        pass
```

Explanation

- `class` → Keyword to define a class
- `ClassName` → Name of the class (use PascalCase)
- `self` → Refers to the current object
- `def` → Keyword to define a method
- `pass` → Placeholder (empty body)

Example (Python)

```
class Student:
    # Constructor (optional but commonly used)
    def __init__(self, name, rollno, marks):
        self.name = name          # attribute 1
        self.rollno = rollno      # attribute 2
        self.marks = marks        # attribute 3

    # Method to display details
    def display(self):
        print("Name   :", self.name)
        print("Roll No:", self.rollno)
        print("Marks  :", self.marks)

# Creating objects (instances) of Student class
s1 = Student("Amit", 101, 85)
s2 = Student("Neha", 102, 92)

# Accessing method
s1.display()
s2.display()
```

Output

```
Name   : Amit
Roll No: 101
Marks  : 85

Name   : Neha
Roll No: 102
Marks  : 92
```

Note:
`self` is a reference to the current object. It must be the first parameter in every method (except static methods).

Benefits of Using Classes

- Promotes code reusability
- Organizes code in a better way
- Improves data security
- Easier to maintain and extend
- Models real-world entities efficiently


1010101010
0101010101
1010101010
0101010101
Unit -II
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Functions: Function Prototype

What is a Function Prototype?

A function prototype (or declaration) tells the compiler about a function's name, return type, and parameters before the actual function is used. It does not contain the body.

Why do we need it?

- ✓ Allows the compiler to check function calls for correctness.
- ✓ Helps in calling a function before its definition.
- ✓ Improves code organization.
- ✓ Useful in large programs and multiple files.

Note:

A prototype ends with a semicolon (;) and has no function body.

Syntax

```
return_type  function_name( parameter_list );
```

Examples:

```
int add(int, int);
void display();
float area(float, float);
```

Example Program

```
#include <iostream>
using namespace std;

// Function Prototypes (Declarations)
int add(int, int);        // prototype of add()
void display();           // prototype of display()

int main() {
    int a = 10, b = 20;
    cout << "Sum = " << add(a, b) << endl;    // function call
    display();                                // function call
    return 0;
}

// Function Definitions

int add(int x, int y) {
    return x + y;
}

void display() {
    cout << "Hello! This is a function." << endl;
}
```

Key Points

- A prototype is also called a function declaration.
- It specifies the return type, function name, and parameters.
- Parameter names are optional in a prototype.
- A prototype ends with a semicolon.
- The actual definition (with body) can appear later in the program.

How it Works

Compiler sees the prototype first → Compiler knows about the function (name, return type, parameters)

↓

Function can be called anywhere in the code (before or after its definition)

Remember

Prototype = Declaration

Definition = Actual Body

Inline Function in C++

What is Inline Function?

An inline function is a function in which the compiler tries to replace the function call with the actual function code.

It is specified using the inline keyword.

Why Inline Functions?

- ✓ Reduces function call overhead
- ✓ Improves program execution speed
- ✓ Useful for small and frequently called functions
- ✓ The compiler decides whether to expand the function or not.

Note

Inline is just a request to the compiler, not a command.
The compiler may ignore it if the function is large or complex.

Syntax

```
inline return_type function_name(parameters)
{
    // function body
}
```

Example

```
#include <iostream>
using namespace std;

// Inline function
inline int add(int a, int b)
{
    return a + b;
}
int main()
{
    int x = 10, y = 20, sum;
    sum = add(x, y);   // function call
    cout << "Sum = " << sum << endl;
    return 0;
}
```

Here, the compiler may replace the call `add(x, y)` with the actual code of `add()`.

How it Works?

1. When the program is compiled, the compiler sees the inline function.
2. At the place of function call, the compiler copies the function body.
3. This eliminates the overhead of function call and return.

Important Points

- Use inline for small functions (1-5 lines).
- It is usually placed in header files.
- Recursive functions cannot be inline.
- Excessive use may increase program size.

Inline functions are the best choice when performance is critical and the function is small and simple.

Function Overloading in C++

What is Function Overloading?

Function overloading is a feature in C++ that allows a class or a program to have more than one function with the same name but different parameters.

Each function performs a similar task but in different ways.

Why Use Function Overloading?

- Improves code readability.
- Helps to use the same function name for different types of data.
- Reduces the need to remember multiple function names.
- Makes the program easy to maintain and understand.

Rules

1. Functions must have the same name.
2. Functions must have different parameter lists (different number, type or order of parameters).
3. Return type may be same or different, but cannot be used alone to overload a function.
4. Function overloading is not possible by changing return type only.

Syntax

```
return_type  function_name(parameter_list_1);
return_type  function_name(parameter_list_2);
...
return_type  function_name(parameter_list_n);
// all have same name but different parameters
```

Example Program

```
#include <iostream>
using namespace std;

// Function Overloading
int add(int a, int b)            // two integers
{
    return a + b;
}
float add(float a, float b)      // two floats
{
    return a + b;
}
int add(int a, int b, int c)     // three integers
{
    return a + b + c;
}
int main()
{
    cout << "add(2, 3) = " << add(2, 3) << endl;         // calls first
    cout << "add(2.5, 3.5) = " << add(2.5, 3.5) << endl; // calls second
    cout << "add(1, 2, 3) = " << add(1, 2, 3) << endl; // calls third
    return 0;
}
```

Output

```
add(2, 3) = 5
add(2.5, 3.5) = 6
add(1, 2, 3) = 6
```

Compiler decides which function to call based on the number and type of arguments.

How It Works?

1. When a function is called, the compiler looks at the name and the arguments passed.
2. It matches the call with the function that has the same name and the matching parameter list.
3. The corresponding function is executed.

Key Points

- Function overloading is also called compile-time polymorphism.
- It improves code flexibility.
- It is also known as static polymorphism or early binding.
- It helps in writing generic functions.

Important Note

Functions cannot be overloaded by changing only the return type.

Example (Invalid):

```
int area(int r);        // Invalid
float area(int r);      // Invalid
// Compiler will give an error
```

Function Overloading allows you to use one function name for different tasks, making your programs cleaner and more efficient.

Constructors: Types of Constructors

Constructors have the same name as the class and have no return type.

A constructor is a special member function of a class that is automatically called when an object of the class is created. It is used to initialize the data members of the class.

Types of Constructors in C++

1. Default Constructor

A default constructor is a constructor that takes no arguments.

It is provided by the compiler if we do not define any constructor.

Example

```
class Point {
   int  x, y;
public:
   Point() {   // default constructor
      x = 0;
      y = 0;
   }
};
```

Usage

```
Point p;   // Default constructor is called
```

2. Parameterized Constructor

A constructor that takes one or more arguments to initialize the data members with user-defined values.

Example

```
class Point {
   int  x, y;
public:
   Point(int a, int b) {
      x = a;
      y = b;
   }
};
```

Usage

```
Point p(10, 20);   // Parameterized
                   constructor is called
```

3. Copy Constructor

A constructor that initializes an object using another object of the same class.

Example

```
class Point {
   int  x, y;
public:
   Point(int a, int b) { x = a; y = b; }
   Point(const Point &p) {  // copy
      x = p.x;
      y = p.y;
   }
};
```

Usage

```
Point p1(5, 10);
Point p2 = p1; // copy constructor is called
```

4. Dynamic Constructor

A constructor that allocates memory dynamically using new operator.

Example

```
class Array {
   int *ptr;
   int size;
public:
   Array(int n) {
      size = n;
      ptr = new int[size];
   }
   ~Array() { delete[] ptr; }
};
```

Usage

```
Array a(5);  // Dynamic constructor is called
```

Key Takeaways

- ✓ Constructors are special member functions used to initialize objects.
- ✓ There are four main types: Default, Parameterized, Copy, and Dynamic constructors.
- ✓ The type of constructor used depends on the way we create the object.

Constructors in C++
Default, Parameterized and Copy Constructor

Constructors are special member functions that are automatically called when an object is created.

A constructor initializes the data members of an object. It has the same name as the class, no return type and is called automatically.

1. Default Constructor

- A constructor that takes no arguments.
- If we do not write any constructor, the compiler provides a default constructor.

Syntax

```
ClassName() {   // no parameters }
```

Example

```
class Student {
    int roll;
    string name;
public:
    // Default Constructor
    Student() {
        roll = 0;
        name = "Unknown";
    }
};
```

Object Created

```
Student s;

s.roll = 0
s.name = "Unknown"
```

Usage

```
Student s;   // Default constructor is called
```

2. Parameterized Constructor

- A constructor that takes one or more arguments.
- It is used to initialize objects with user-defined values.

Syntax

```
ClassName(type1 p1, type2 p2, ..) {
    // initialization
}
```

Example

```
class Student {
    int roll;
    string name;
public:
    // Parameterized Constructor
    Student(int r, string n) {
        roll = r;
        name = n;
    }
};
```

Object Created

```
Student s(101,
        "Shahid");

s.roll = 101
s.name = "Shahid"
```

Usage

```
Student s(101, "Shahid");   // Parameterized constructor is called
```

3. Copy Constructor

- A constructor that initializes an object using another object of the same class.
- It is used to create a new object as a copy of an existing object.

Syntax

```
ClassName(const ClassName &obj) {
    // initialization
}
```

Example

```
class Student {
    int roll;
    string name;
public:
    Student(int r, string n) {   // Parameterized
        roll = r;
        name = n;
    }
    // Copy Constructor
    Student(const Student &s) {
        roll = s.roll;
        name = s.name;
    }
};
```

Objects

s1
roll = 101
name = "Shahid"

Copy

s2
roll = 101
name = "Shahid"

Usage

```
Student s1(101, "Shahid");   // Parameterized constructor called
Student s2 = s1;             // Copy constructor called
```

Key Points

- ✓ Constructor name = Class name
- ✓ No return type (not even void)
- ✓ Constructors are called automatically
- ✓ Used to initialize objects

Summary

- ◆ Default Constructor → takes no arguments, sets default values.
- ◆ Parameterized Constructor → takes arguments, sets values as per the arguments.
- ◆ Copy Constructor → creates a new object as a copy of an existing object.

Inheritance in C++

1. DEFINITION

Inheritance is one of the key features of OOPs in C++.
It is the process by which a new class (derived class / subclass) acquires the properties (data members) and behaviors (member functions) of an existing class (base class / superclass).

Base Class	Inheritance	Derived Class
• Data members • Member functions	⟹	• Inherited members • Additional members

Purpose: Reusability of code, Extensibility, Easier maintenance.

Example of Inheritance (Single)

```
#include <iostream>
using namespace std;

class Animal {
public:
  void eat() { cout << "Eating..." << endl; }
};

class Dog : public Animal {       // Dog inherits from Animal
public:
   void bark() { cout << "Barking..." << endl; }
};

int main() {
    Dog d;          // Object of derived class
    d.eat();        // Inherited member
    d.bark();       // Own member
    return 0;
}
```

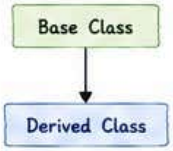
Output

```
Eating...
Barking...
```

2. TYPES OF INHERITANCE

1. Single Inheritance

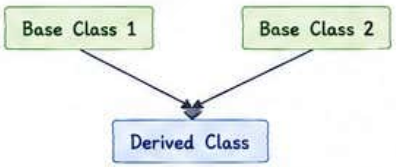
One derived class inherits from one base class.




```
Example:
class A { /* ... */ };
class B : public A { /* ... */ };
// B inherits from A
```

2. Multiple Inheritance

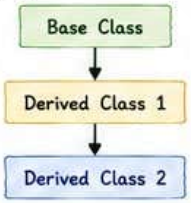
One derived class inherits from more than one base class.




```
Example:
class A { /* ... */ };
class B { /* ... */ };
class C : public A, public B { /* ... */ };
// C inherits from A and B
```

3. Multilevel Inheritance

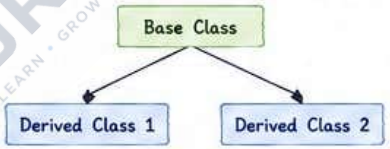
A derived class is created from another derived class.




```
Example:
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
// C inherits from B, B inherits from A
```

4. Hierarchical Inheritance

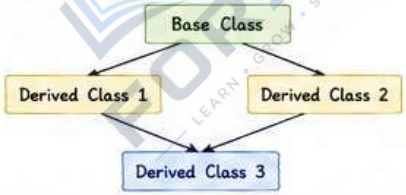
More than one derived class inherits from the same base class.




```
Example:
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
// B and C inherit from A
```

5. Hybrid Inheritance

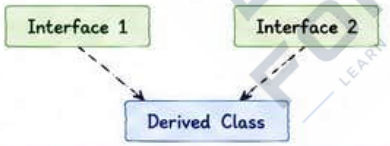
Combination of two or more types of inheritance.




```
Example:
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };
// Combination of multiple and multilevel
```

6. Multiple (by Interfaces) / Realization Inheritance (in C++)

Achieved using Interfaces (abstract classes) in C++.




```
Example:
class I1 { public: virtual void show() = 0; };
class I2 { public: virtual void display() = 0; };
class D : public I1, public I2 {
   public: void show(); void display();
};
// D implements both I1 and I2
```

Key Points

- Inheritance promotes code reusability.
- A derived class can add new data members and member functions.
- It represents an "is-a" relationship.
- In C++, inheritance is indicated using ':'.

Access Specifier in Inheritance

Mode of Inheritance	public members of base class	protected members	private members
public Inheritance	public in derived class	protected in derived class	Not accessible
protected Inheritance	protected in derived class	protected in derived class	Not accessible
private Inheritance	private in derived class	private in derived class	Not accessible

Note

- Use inheritance carefully to model real-world relationships.
- Overuse may lead to complex and hard-to-maintain code.

Inheritance in C++

Inheritance is the process by which one class acquires the properties (data members) and behaviors (member functions) of another class.

1. DEFINITION

Inheritance is one of the key features of Object-Oriented Programming (OOP). It allows us to create a new class (derived class / subclass) from an existing class (base class / superclass). The derived class inherits the data members and member functions of the base class.

Base Class (Parent Class)
- Data Members
- Member Functions

Inheritance →

Derived Class (Child Class)
- Inherits members of base class
- Can have its own additional members

Purpose: Code reusability, reduces redundancy, easy to maintain and extend.

Key Points

- Inheritance promotes code resuability.
- It establishes an "IS-A" relationship.
- The derived class can use members of the base class.
- A derived class can have its own members in addition to inherited members.
- In C++, inheritance is achieved using the ':' symbol.

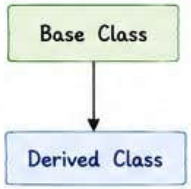
Note: Inheritance makes the program modular, easy to extend and maintain.

2. TYPES OF INHERITANCE

A. SINGLE INHERITANCE

In single inheritance, a derived class inherits from only one base class.

Diagram:



Example:

```cpp
#include <iostream>
using namespace std;

class A {                    // Base class
public:
   int x;
   void showA() { cout << "Value of x: " << x << endl; }
};

class B : public A {         // Derived class
public:
   int y;
   void showB() { cout << "Value of y: " << y << endl; }
};

int main() {
   B obj;                    // Object of derived class
   obj.x = 10;               // Inherited member of A
   obj.y = 20;               // Own member of B
   obj.showA();              // Access base class function
   obj.showB();              // Access derived class function
   return 0;
}
```

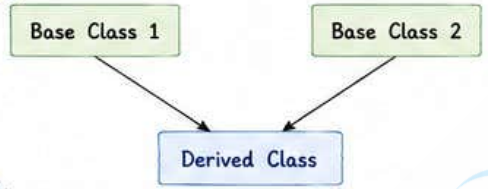
Output:

```
Value of x: 10
Value of y: 20
```

B. MULTIPLE INHERITANCE

In multiple inheritance, a derived class inherits from more than one base class.

Diagram:



Example:

```cpp
#include <iostream>
using namespace std;

class A {                    // Base class 1
public:
   int a;
   void showA() { cout << "Value of a: " << a << endl; }
};

class B {                    // Base class 2
public:
   int b;
   void showB() { cout << "Value of b: " << b << endl; }
};

class C : public A, public B {      // Derived class
public:
   int c;
   void showC() { cout << "Value of c: " << c << endl; }
};

int main() {
   C obj;                    // Object of derived class
   obj.a = 10;               // From base class A
   obj.b = 20;               // From base class B
   obj.c = 30;               // Own member of C
   obj.showA();
   obj.showB();
   obj.showC();
   return 0;
}
```

Output:

```
Value of a: 10
Value of b: 20
Value of c: 30
```

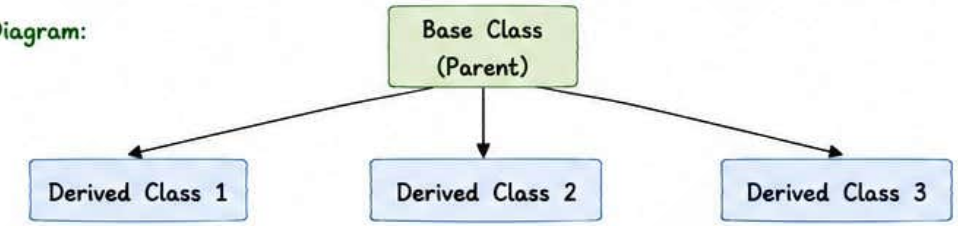
Inheritance in C++

Inheritance is the mechanism in OOP by which a new class (derived class) acquires the properties and behaviors of an existing class (base class).

1. HIERARCHICAL INHERITANCE

In hierarchical inheritance, more than one derived class inherits from the same base class.

Diagram:



Example (C++ Code):

```cpp
#include <iostream>
using namespace std;

class Shape {                    // Base class
public:
    void area() { cout << "This is a shape.\n"; }
};

class Circle : public Shape {    // Derived class 1
public:
    void area() { cout << "Area of Circle\n"; }
};

class Rectangle : public Shape {   // Derived class 2
public:
    void area() { cout << "Area of Rectangle\n"; }
};

class Triangle : public Shape {    // Derived class 3
public:
    void area() { cout << "Area of Triangle\n"; }
};

int main() {
    Circle c;  Rectangle r;  Triangle t;
    c.area();     // Output: Area of Circle
    r.area();     // Output: Area of Rectangle
    t.area();     // Output: Area of Triangle
    return 0;
}
```

Key Points

- One base class is shared by multiple derived classes.
- Each derived class inherits independently.
- Changes in base class are reflected in all derived classes.

Syntax

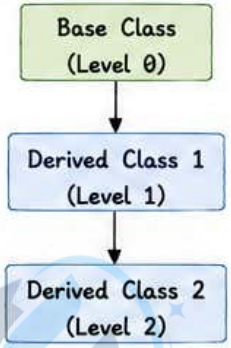
```cpp
class Derived1 : access_specifier Base
{
   // body
};

class Derived2 : access_specifier Base

   // body
};
 ...
```

access_specifier can be public, protected or private.

2. MULTILEVEL INHERITANCE

In multilevel inheritance, a derived class is created from another derived class.

Diagram:



Key Points

- Inheritance occurs in a chain.
- Each derived class becomes the base class for the next class.
- Features of base class are passed to derived class1 and then to derived class2.

Example (C++ Code):

```cpp
#include <iostream>
using namespace std;

class Animal {                  // Base class
public:
    void eat() { cout << "Animal eats food.\n"; }
};

class Mammal : public Animal {  // Derived class 1
public:
    void walk() { cout << "Mammal walks.\n"; }
};

class Dog : public Mammal {     // Derived class 2
public:
    void bark() { cout << "Dog barks.\n"; }
};

int main() {
    Dog d;
    d.eat();     // from Animal
    d.walk();    // from Mammal
    d.bark();    // from Dog
    return 0;
}
```

Syntax

```cpp
class Base {
    // body
};

class Derived1 : access_specifier Base {
    // body
};

class Derived2 : access_specifier Derived1 {
    // body
};
```

Output:

```
Animal eats food.
Mammal walks.
Dog barks.
```

Both hierarchical and multilevel inheritance promote code reusability, reduce redundancy and support easier maintenance.

Inheritance in C++

Hybrid Inheritance is a **combination** of **two or more** types of inheritance.

1. DEFINITION

Hybrid Inheritance in C++ is achieved when a combination of two or more types of inheritance (such as multiple, multilevel, hierarchical, etc.) is used in the same program.

2. KEY FEATURES

- Combines multiple inheritance patterns.
- Provides flexibility and reusability.
- Helps in building complex and real-world relationships.
- Supports code reusability and reduces redundancy.

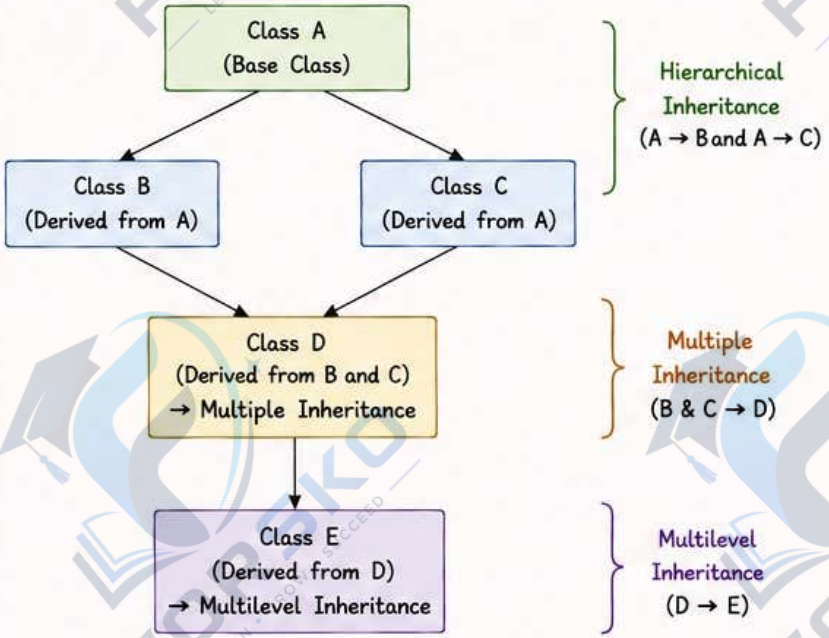
3. WHY USE HYBRID INHERITANCE?

Reusability
Code of common behavior can be used in multiple classes.

Flexibility
Allows combination of different inheritance types.

Real-world Modeling
Better representation of real-world entities and relationships.

4. HYBRID INHERITANCE DIAGRAM (Example)



This is an example of Hybrid Inheritance combining:
Hierarchical + Multiple + Multilevel

5. EXAMPLE IN C++

```cpp
#include <iostream>
using namespace std;

class A {
public:
    void showA() { cout << "Function of Class A\n"; }
};

class B : public A {          // Derived from A
public:
    void showB() { cout << "Function of Class B\n"; }
};

class C : public A {          // Derived from A
public:
    void showC() { cout << "Function of Class C\n"; }
};

class D : public B, public C {    // Multiple Inheritance
public:
    void showD() { cout << "Function of Class D\n"; }
};

class E : public D {              // Multilevel Inheritance
public:
    void showE() { cout << "Function of Class E\n"; }
};

int main() {
    E obj;
    obj.showA();   // from A
    obj.showB();   // from B
    obj.showC();   // from C
    obj.showD();   // from D
    obj.showE();   // from E
    return 0;
}
```

OUTPUT

```
Function of Class A
Function of Class B
Function of Class C
Function of Class D
Function of Class E
```

6. ADVANTAGES

- Enhances code reusability.
- Provides a clear and logical structure.
- Helps in managing large and complex programs.
- Improves maintainability and scalability.

7. DISADVANTAGES

- Complexity increases.
- Harder to understand and maintain.
- More chances of ambiguity (naming conflicts).
- Requires good design and planning.

8. NOTE

Hybrid Inheritance is rarely used in practice because it can make the program complex. Use it only when necessary and when it improves the design of the system.

Hybrid Inheritance = Combination of different inheritance types to achieve maximum code reusability and flexibility.

Visibility Modes in C++

Visibility Modes (Access Specifiers) in C++ control the accessibility of class members. They are used to implement Encapsulation (data hiding).

1. DEFINITION

Visibility modes determine from where a class member (data member or member function) can be accessed.

2. THE THREE VISIBILITY MODES

public
Accessible from anywhere (inside class, outside class, through object, through derived class).

protected
Accessible inside class and its derived classes.
Not accessible outside the class.

private
Accessible only inside the class.
Not accessible outside the class or in derived classes.

3. COMPARISON OF VISIBILITY MODES

Aspect	public	protected	private
Access within same class	✓ Yes	✓ Yes	✓ Yes
Access from outside class (using object)	✓ Yes	✗ No	✗ No
Access in derived class	✓ Yes	✓ Yes	✗ No
Default in class	No (must be written)	No (must be written)	Yes (default)
Purpose	To provide interface or allow access to all.	To allow access to derived classes, but not outside.	To hide data and prevent direct access from outside.

4. EXAMPLE

```
#include <iostream>
using namespace std;

class Demo {
private:
    int a;              // private member
protected:
    int b;              // protected member
public:
    int c;              // public member
    void setValues() {
        a = 10;
        b = 20;
        c = 30;
    }
};

// Derived class
class Derived : public Demo {
public:
    void show() {
        cout << a << endl;  // Error: a is private
        cout << b << endl;  // OK: b is protected
        cout << c << endl;  // OK: c is public
    }
};

int main() {
    Demo d;
    d.c = 30;       // OK: public
    // d.b = 20;    // Error: protected
    // d.a = 10;    // Error: private
    return 0;
}
}
```

OUTPUT

```
20
30
```

5. ACCESS EXAMPLE

public

```
// Accessible everywhere
obj.c;
```

Class → Object → Anywhere

protected

```
// Accessible in class and derived classes
obj.b;
```

Class → Derived Class → Object

private

```
// Accessible only inside class
a;
```

Class → (Inside Class only)

6. KEY POINTS

- private is the most restrictive.
- public is the least restrictive.
- protected provides controlled access for inheritance.
- These modes help in data hiding and achieving encapsulation.
- In a class, members are private by default if no specifier is written.

7. BEST PRACTICES

- Keep data members private.
- Provide public member functions (getters/setters) to access or modify private data.
- Use protected when you want derived classes to access, but not outside users.

Remember: Use the most restrictive visibility that makes sense. This improves security, reduces bugs and makes code easy to maintain.

Unit - III


1010101010
0101010101
1010101010
0101010101
</>
</>
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
10110
01001
10101
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Introduction:
History of Python Programming Language

Python is a high-level, interpreted, general-purpose programming language.
Its design philosophy emphasizes code readability with the use of significant indentation.

Late 1980s
The Beginning

Guido van Rossum, a programmer from the Netherlands, started working on Python.

1991
First Release

Python 0.9.0 was released in February 1991. It included basic features like functions, modules, and exceptions.

1994
Python 1.0

Python 1.0 was released in January 1994 with new features like lambda, map, filter, and reduce.

2000
Python 2.0

Python 2.0 introduced new features such as list comprehensions and garbage collection improvements.

2008
Python 3.0

Python 3.0 was released in December 2008 with major changes that made it not fully backward compatible.

2010s
Rapid Growth

Python gained immense popularity in areas like Web Development, Data Science, AI, Automation, and more.

2020s and Beyond
The Future

Python continues to evolve with new features, better performance, and a strong global community.

Why Python Was Created?

- Guido wanted a language that was easy to read and write.
- Inspired by languages like ABC, Modula-3, and C.
- Aimed to reduce the complexity of programming.

Key Highlights

- Named after the British comedy group "Monty Python".
- Open source and community-driven.
- Emphasizes readability and simplicity.
- *"Beautiful is better than ugly." – The Zen of Python*

Today, Python is

- One of the most popular programming languages
- Used in countless industries and applications
- Loved by beginners and professionals alike

> *The mission of Python is to be a constructive force in the world.*
> – Guido van Rossum

python™

Official Website
https://www.python.org

Thrust Areas of Python

Python is a versatile programming language that powers innovation across a wide range of domains.
Its simplicity, rich libraries, and strong community make it a top choice in today's world.

1 Web Development

- Build dynamic and scalable websites and web applications.
- Frameworks: Django, Flask, FastAPI
- Used by top companies like Instagram, Pinterest, YouTube

2 Data Science & Analytics

- Analyze, visualize, and extract insights from data.
- Libraries: NumPy, Pandas, Matplotlib, Seaborn
- Widely used in research, business intelligence, and decision making

3 Artificial Intelligence & Machine Learning

- Build intelligent systems that learn and improve from data.
- Libraries: TensorFlow, PyTorch, scikit-learn, Keras
- Applications in NLP, computer vision, robotics, and more

4 Automation & Scripting

- Automate repetitive tasks and streamline workflows.
- Used in system admin, file handling, web scraping, testing, and more.
- Boost productivity with simple and powerful scripts

5 Data Engineering & Big Data

- Build and manage large-scale data pipelines.
- Tools: PySpark, Airflow, Dask
- Handle massive data efficiently for analytics and ML applications

6 Scientific Computing & Research

- Perform complex mathematical computations and simulations.
- Libraries: SciPy, SymPy, NumPy
- Used in engineering, physics, chemistry, and academia

7 Desktop GUI Applications

- Build cross-platform desktop applications.
- Frameworks: Tkinter, PyQt, Kivy
- Suitable for tools, utilities, and enterprise applications

8 Game Development

- Create games using Python's simple syntax and libraries.
- Libraries/Engines: Pygame, Panda3D
- Great for 2D games, prototypes, and learning game logic

9 Internet of Things (IoT)

- Python connects hardware and software seamlessly.
- Libraries: MicroPython, RPi.GPIO, PySerial
- Used in smart devices, automation, sensors, and embedded systems

10 Education & Learning

- Beginner-friendly syntax makes it ideal for learning programming.
- Widely used in schools, colleges, and online courses.
- Builds a strong foundation for all tech fields

Why Python Leads?

- ✓ Easy to learn and use
- ✓ Cross-platform and open source
- ✓ Huge ecosystem of libraries
- ✓ Strong community support

> *Python empowers ideas, drives innovation, and builds the future.*

Python Everywhere!

From startups to space missions, Python is the language of choice across industries and innovations.

python™

Parts of Python Programming Language: Identifiers, Keywords

Like every programming language, Python has building blocks.
Two important building blocks are **Identifiers** and **Keywords**.

IDENTIFIERS

Identifiers are names given to entities like variables, functions, classes, objects, modules, etc.

Rules for Identifiers

- Must start with a letter (A–Z or a–z) or underscore (_).
- After the first character, you can use letters, digits (0–9) or underscores (_).
- Identifiers are case-sensitive (name, Name and NAME are different).
- Cannot be a Python keyword.
- No special characters allowed (except _).

Examples

Valid Identifiers

`total`	`_count`	`value2`
`student_name`	`getData`	`num_1`
`A1`	`_temp`	`myVariable`

Invalid Identifiers

`2value`	`student-name`	`total$`
`class`	`my var`	`@data`
`if`	`temp!`	`1st`

Example in Code

```
# Valid identifiers
student_name = "Alice"
total_marks = 450
_count = 1
print(student_name, total_marks, _count)
```

Good Practice

Use meaningful and descriptive names so your code is easy to read and understand.

KEYWORDS

Keywords are reserved words in Python that have a special meaning.

Characteristics

- Keywords are part of Python syntax.
- They cannot be used as identifiers (variable names, function names, etc.).
- All keywords are written in lowercase.

List of Python Keywords (Python 3.x)

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Example (Invalid Use)

```
# Invalid: using keyword as variable name
if = 10          # ✗ SyntaxError
for = 5          # ✗ SyntaxError
```

Note: You cannot use keywords as names for variables, functions, classes, or any other identifiers.

Check All Keywords

You can get the list of all keywords in Python using the keyword module.

```
import keyword
print(keyword.kwlist)
```

Identifiers vs Keywords

Feature	Identifiers	Keywords
Purpose	User-defined names	Reserved by Python
Examples	`name, total, value1`	`if, for, while`
Can we change?	Yes	No
Can be used as names?	Yes	No

Tips

- Choose meaningful identifiers.
- Follow naming conventions: use lowercase with underscores (e.g., `total_marks`).
- Avoid using Python keywords as identifiers.

Quick Check

Which of the following are valid identifiers?

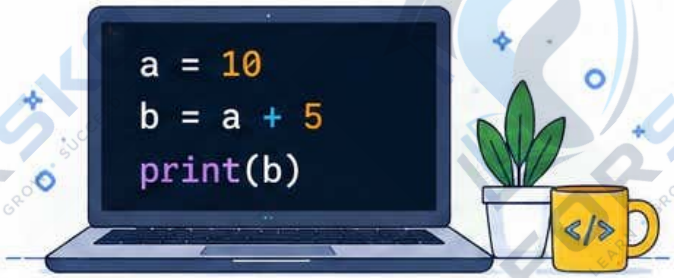
- ✓ student1
- ✓ _value
- ✓ total_marks
- ✗ 2name
- ✗ for
- ✗ my-name

Learn the rules, follow the style, write better Python programs!

Identifiers are your names. Keywords are Python's words. Use them correctly to write powerful programs!

Python Basics: Statements and Expressions

Every Python program is made up of **statements**.
Statements are built from **expressions**.



STATEMENTS

A statement is a complete instruction that Python can execute. It **performs an action**.

Examples of Statements

Statement Type	Example	What it does
Assignment Statement	`x = 25`	Assigns the value 25 to variable x
Print Statement	`print(x)`	Displays the value of x
Input Statement	`name = input("Enter name: ")`	Takes input from the user
Conditional Statement	`if x > 0:` `  print("Positive")`	Executes block if condition is true
Loop Statement	`for i in range(3):` `  print(i)`	Repeats a block of code
Function Call Statement	`greet("Hello")`	Calls a function

Key Point: A statement stands on its own as a complete instruction.

EXPRESSIONS

An expression is a combination of values, variables, operators, and function calls that **evaluates to a value.**

Examples of Expressions

Expression	Evaluates to
`10`	10
`x`	value of x
`x + 5`	sum of x and 5
`(a * b) / 2`	result of a * b, then divided by 2
`len("Python")`	6
`x > 0`	True or False

Key Point: An expression always produces a value.

RELATIONSHIP

Expressions Produce a value → **Used in** → **Statements** Perform an action

Example

```
x = 10 + 5      # 10 + 5 is an expression that evaluates to 15
print(x)        # print(x) is a statement that displays the value
```

More Examples

```
total = price * qty      # price * qty → expression
if total > 100:          # total > 100 → expression used in a statement
    print("Discount")    # print(...) → statement
```

DETAILED EXAMPLES

1. Assignment Statement

```
a = 7 + 3     # 7 + 3 is an expression (evaluates to 10)
b = a * 2     # a * 2 is an expression (evaluates to 20)
print(b)      # print(b) is a statement
```

2. Conditional Statement

```
x = 15                  # statement
if x % 2 == 0:          # condition is an expression
    print("Even")       # statement
else:
    print("Odd")        # statement
```

3. Loop Statement

```
for i in range(3):      # statement
    print(i)            # print(i) is a statement
```

4. Function Call Statement

```
def add(a, b):          # def is a statement
    return a + b        # a + b is an expression

result = add(4, 6)      # function call statement
print(result)           # print is a statement
```

COMMON MISTAKES

Mistake 1: Writing an expression by itself

```
10 + 5          # This is an expression, but not a statement
                # Nothing will be displayed or stored
```

Mistake 2: Forgetting indentation in block statements

```
if x > 0:
print("Positive")       # IndentationError
```

Mistake 3: Using keywords as identifiers

```
for = 10                # SyntaxError: invalid syntax
```

QUICK SUMMARY

- Expressions evaluate to a value.
- Statements perform an action.
- Expressions are used inside statements.
- A program is a sequence of statements.

Think of it this way:

Expression `2 + 3` (Produces a value) → **Used in** → Statement `x = 2 + 3` (Performs an action)

Remember: Expressions give values; Statements do something with those values.

Master statements and expressions – build the foundation of every Python program!

Python Basics: Variables, Operators

Variables are used to store data. Operators are used to perform operations on data and variables.

VARIABLES

A variable is a name (identifier) that refers to a value stored in computer memory. The value can change during program execution.

1. Creating Variables

- No need to declare type in Python.
- Use meaningful names.
- Case-sensitive.

Examples:

```
x = 10
name = "Alice"
pi = 3.14
is_valid = True
```

2. Rules for Variables

- Must start with a letter (A–Z or a–z) or underscore (_).
- After the first character, can contain letters, digits (0–9) or underscores (_).
- Cannot be a Python keyword.
- Case-sensitive (**age**, **Age** and AGE are different).
- No spaces or special characters allowed.

3. Updating Variables

The value stored in a variable can be changed.

```
x = 5
print(x)      # 5
x = x + 10
print(x)      # 15
```

4. Multiple Assignment

You can assign multiple variables in one statement.

```
a, b, c = 1, 2, 3
x = y = z = 0
```

5. Data Types (Examples)

Type	Example	Description
int	x = 10	Integer (whole number)
float	p = 3.14	Decimal number
str	name = "Bob"	Text (string)
bool	flag = True	Boolean (True or False)

Tips

- Choose meaningful variable names.
- Follow **snake_case** style (e.g., **total_marks**).
- Avoid using keywords as variable names.

Common Mistakes

```
1variable = 10      # Invalid: cannot start with digit
total marks = 100   # Invalid: spaces not allowed
for = 5             # Invalid: 'for' is a keyword
```

OPERATORS

Operators are special symbols that perform operations on one, two or more operands (values/variables).

TYPES OF OPERATORS

1. Arithmetic Operators

Operator	Meaning	Example	Result
+	Addition	7 + 3	10
-	Subtraction	7 - 3	4
*	Multiplication	7 * 3	21
/	Division (float)	7 / 3	2.333...
//	Floor Division	7 // 3	2
%	Modulus (Remainder)	7 % 3	1
**	Exponentiation	7 ** 3	343

2. Comparison (Relational) Operators

Operator	Meaning	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal to	5 >= 5	True
<=	Less than or equal to	5 <= 3	False

3. Assignment Operators

Operator	Meaning	Example	Same as
=	Assign	x = 10	x = 10
+=	Add and assign	x += 5	x = x + 5
-=	Subtract and assign	x -= 3	x = x - 3
*=	Multiply and assign	x *= 2	x = x * 2
/=	Divide and assign	x /= 4	x = x / 4
%=	Modulus and assign	x %= 3	x = x % 3
**=	Exponent and assign	x **= 2	x = x ** 2
//=	Floor divide and assign	x //= 2	x = x // 2

4. Logical Operators

Operator	Meaning	Example	Result
and	True if both are True	(5 > 3) and (2 < 4)	True
or	True if at least one is True	(5 > 3) or (2 > 4)	True
not	Reverse the result	not (5 > 3)	False

5. Membership Operators

Operator	Meaning	Example	Result
in	True if value exists in sequence	'a' in 'apple'	True
not in	True if value does not exist in sequence	'z' not in 'apple'	True

6. Identity Operators

Operator	Meaning	Example	Result
is	True if both operands refer to same object	a is b	True/False
is not	True if both operands refer to different objects	a is not b	True/False

Operator Precedence (Highest to Lowest)

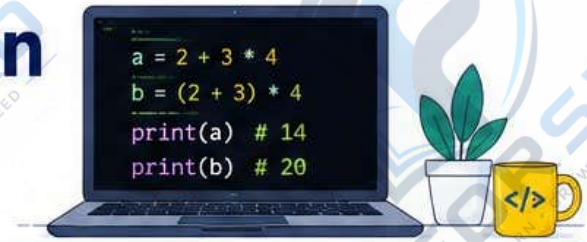
() → ** → unary +, not → *, /, //, % → +,- → comparison operators → not → and → or

Use parentheses () to change the order.

Variables store data. Operators work on data. Together they build powerful Python programs!

Precedence and Associativity in Python

When an expression contains multiple operators, the order in which they are evaluated is determined by **Precedence** and **Associativity**.



PRECEDENCE

Precedence decides which operator is evaluated first in an expression. Operators with higher precedence are evaluated before operators with lower precedence.

ASSOCIATIVITY

Associativity decides the order of evaluation when two or more operators of the same precedence appear in an expression. It can be Left-to-Right or Right-to-Left.

Operator Precedence in Python (Highest to Lowest)

Precedence Level	Operators	Description	Example	Evaluates As
1	`( )`	Parentheses (grouping)	`(2 + 3) * 4`	`5 * 4 = 20`
2	`**`	Exponentiation	`2 ** 3 ** 2`	`2 ** (3 ** 2) = 512`
3	`+x  -x  ~x`	Unary plus, minus, bitwise NOT	`-5 + 2`	`(-5) + 2 = -3`
4	`*  /  //  %`	Multiplication, Division, Floor Division, Modulus	`7 * 3 / 2`	`21 / 2 = 10.5`
5	`+  -`	Addition, Subtraction	`7 + 3 - 2`	`10 - 2 = 8`
6	`<<  >>`	Bitwise Left shift, Right shift	`8 << 1`	`16`
7	`&`	Bitwise AND	`5 & 3`	`1`
8	`^`	Bitwise XOR	`5 ^ 3`	`6`
9	`|`	Bitwise OR	`5 | 3`	`7`
10	`==  !=  >  <  >=  <=`	Comparisons	`5 > 3`	`True`
11	`not`	Logical NOT	`not True`	`False`
12	`and`	Logical AND	`True and False`	`False`
13	`or`	Logical OR	`True or False`	`True`
14	`if - else`	Conditional (ternary) expression	`x if c else y`	Depends on c
15	`=  +=  -=  *=  /=  //=  %=  **=  &=  |=  ^=  <<=  >>=`	Assignment operators	`x += 1`	`x = x + 1`

Note: If operators have different precedence, the one with higher precedence is evaluated first. If they have the same precedence, associativity decides the order.

Associativity in Python

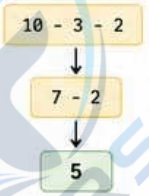
Left-to-Right (Most operators)

When operators of the same precedence appear, evaluation is done from left to right.

Examples: `+`, `-`, `*`, `/`, `//`, `%`, `&`, `|`, `^`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `and`, `or`

Example 1:

```
10 - 3 - 2
=(10 - 3) - 2
= 7 - 2
= 5
```



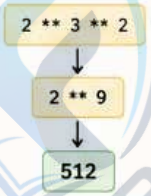
Right-to-Left (Few operators)

When operators of the same precedence appear, evaluation is done from right to left.

Examples: `**`, `=` (and other assignment operators)

Example 2 (Right-to-Left for `**`):

```
2 ** 3 ** 2
= 2 ** (3 ** 2)
= 2 ** 9
= 512
```

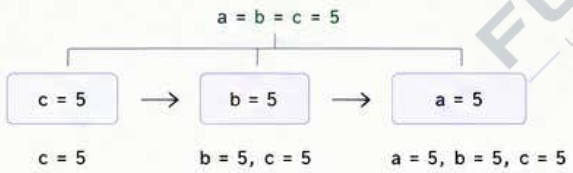


Example 3 (Assignment – Right-to-Left)

```
a = b = c = 5
```

Evaluates as: `a = (b = (c = 5))`

First `c = 5`, then `b = 5`, then `a = 5`.



Let's Evaluate Some Expressions

Example A

```
expr = 2 + 3 * 4 - 5 / (1 + 1)
Step 1: ()
    2 + 3 * 4 - 5 / 2
Step 2: * and /
    2 + 12 - 2.5
Step 3: + and - (Left to Right)
    14 - 2.5 = 11.5
```

Result: 11.5

Example B

```
expr = 2 ** 3 ** 2 + 10
Step 1: ** (Right to Left)
    2 ** (3 ** 2) + 10
    = 2 ** 9 + 10
Step 2: +
    = 512 + 10 = 522
```

Result: 522

Key Takeaways

- Parentheses `()` have the highest precedence.
- Exponentiation `**` is right-associative.
- Most operators are left-associative.
- When in doubt, use parentheses to make the order clear.

Quick Tips

- Operators with higher precedence are evaluated first.
- If precedence is same, associativity decides.
- Use parentheses to avoid confusion and improve readability.

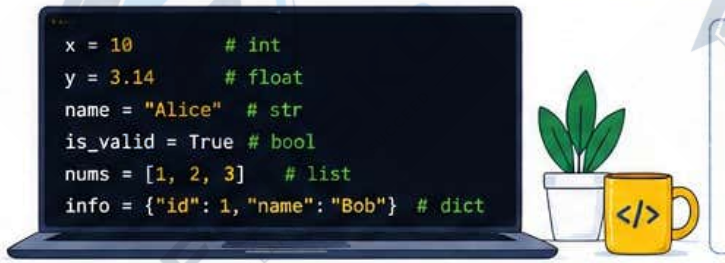
Precedence Order (Highest → Lowest)

Understanding Precedence and Associativity helps you predict how Python evaluates expressions accurately.

Write clear expressions. Write better Python!

Data Types in Python

A data type specifies the type of value a variable can hold and the operations that can be performed on that value.



Remember

The type of data determines what values can be stored and what operations can be performed.

Built-in Data Types in Python

1 Numeric (int, float, complex)

Used to store numbers.

- `int` → Whole numbers
- `float` → Decimal numbers
- `complex` → Complex numbers

Examples

```
a = 10          # int
b = 3.14        # float
c = 2 + 3j      # complex
```

Type: int, float, complex

Properties

- int has unlimited precision.
- float follows double precision.
- complex is written as `a + bj`.

2 String (str)

Used to store sequence of characters (text).

Examples

```
s1 = "Hello"
s2 = 'Python'
s3 = "123"
```

Type: str

Properties

- Enclosed in single, double or triple quotes.
- Immutable (cannot be changed).
- Supports indexing and slicing.

3 Boolean (bool)

Used to store truth values.

Examples

```
b1 = True
b2 = False
```

Type: bool

Properties

- Two values only: `True` or `False`.
- Used in conditions and logical operations.

4 List (list)

Used to store ordered collection of items. Mutable (can change).

Examples

```
nums = [1, 2, 3, 4]
mix = [10, "Hi", 3.5, True]
empty = []
```

Type: list

Properties

- Ordered, mutable.
- Allows duplicate values.
- Supports indexing, slicing, `append()`, `remove()`, etc.

5 Tuple (tuple)

Used to store ordered collection of items. Immutable (cannot change).

Examples

```
t1 = (1, 2, 3)
t2 = (10, "Hi", 3.5)
empty = ()
```

Type: tuple

Properties

- Ordered, immutable.
- Allows duplicate values.
- Supports indexing and slicing.
- Faster and safer than lists.

6 Dictionary (dict)

Used to store key-value pairs. Mutable.

Examples

```
student = {"id": 1,
           "name": "Alice",
           "marks": 95}
```

Type: dict

Properties

- Unordered (in versions < 3.7).
- Keys must be unique and immutable (int, str, tuple, etc.).
- Access values using keys.

7 Check Data Type

Use the `type()` function.

```
type(10)          # <class 'int'>
type(3.14)        # <class 'float'>
type("Hello")     # <class 'str'>
type(True)        # <class 'bool'>
type([1,2,3])     # <class 'list'>
type((1,2,3))     # <class 'tuple'>
type({"a"::1))    # <class 'dict'>
```

8 Type Casting (Conversion)

Convert one data type to another using built-in functions.

Function	Description	Example
`int(x)`	Converts x to integer	`int(3.9)` → 3
`float(x)`	Converts x to float	`float("2.5")` → 2.5
`str(x)`	Converts x to string	`str(100)` → '100'
`list(x)`	Converts x (iterable) to list	`list((1,2))` → [1, 2]
`tuple(x)`	Converts x (iterable) to tuple	`tuple([1,2])` → (1, 2)
`dict(x)`	Converts key-value iterable to dict	`dict([("a",1)])` → {'a': 1}
`bool(x)`	Converts x to boolean	`bool(0)` → False, `bool(5)` → True

Common Mutable vs Immutable Types

Mutable (can be changed)

list, dict, set

Immutable (cannot be changed)

int, float, bool, str, tuple, complex

Trying to change an immutable object creates a new object.

Key Points

- Choosing the correct data type makes your program efficient and bug-free.
- Use list for changeable sequences and tuple for fixed sequences.
- Use dict to store data in key-value pairs for fast access.
- Python is dynamically typed – you do not need to declare the type of a variable.

Understanding data types is the foundation of writing correct and efficient Python programs.

Indentation and Comments in Python

Python uses indentation to define blocks of code.
Comments are used to explain code and make it more readable.

Remember
Good indentation avoids errors.
Good comments improve clarity.

1. INDENTATION

Indentation is the spacing at the beginning of a line.
It tells Python which statements belong to a block.

Rules of Indentation

- Use consistent indentation throughout the program. (Recommended: 4 spaces per indentation level)
- All statements inside a block must have the same indentation level.
- A new block starts after a colon (:)
- Python does not allow mixing spaces and tabs. Choose one and be consistent.

Example: Correct Indentation

```
1  x = 10
2  if x > 5:
3      print("x is greater than 5")      → Block 1
4  else:
5      print("x is 5 or less")           → Block 2
6  print("This is outside the if block") → Outside blocks
```

Example: Incorrect Indentation (causes IndentationError)

```
1  x = 10
2  if x > 5:
3     print("x is greater than 5")
4  else:
5     print("x is 5 or less")
6  print("This will cause an error")
```

✖ Problem
The `print()` statements are not indented correctly.
Python will show:
`IndentationError`

Common Indentation Situations

Situation	What Happens
Missing indentation after :	IndentationError
Extra indentation without reason	IndentationError
Inconsistent use of spaces and tabs	TabError
Correct indentation	Program runs fine

Best Practices

- Use 4 spaces per indentation level (most common standard).
- Do not mix tabs and spaces.
- Use meaningful block structure to improve readability.
- Use your editor settings to show indentation guides.

Visualizing Blocks

```
1  if True:                    → Level 0
2      print("Level 1")        → Level 1
3      if True:
4          print("Level 2")    → Level 2
5      print("Back to Level 1") → Back to Level 1
6  print("Level 0")            → Back to Level 0
```

2. COMMENTS

Comments are ignored by Python interpreter.
They are for humans, not for computers!

Types of Comments

1. Single-line Comments

Start with #. The rest of the line is ignored.

```
x = 10          # This is a comment
y = 20          # Another comment
print(x + y)    # Add x and y
```

2. Multi-line Comments

Use triple quotes ''' or """.
Everything inside is ignored.

```
'''
This is a multi-line comment.
It can span multiple lines.
Useful for long explanations.
'''
print("Hello!")
```

Where to Use Comments?

- To explain what the code does
- To explain why a particular approach is used
- To leave notes for yourself or others
- To temporarily disable code (use with caution)

Example with Comments

```
1  # Program to calculate area of rectangle
2  length = 10       # length of the rectangle
3  width = 5         # width of the rectangle
4
5  # Calculate area
6  area = length * width      # area = l * w
7
8  print("Area =", area)      # display the result
```

Common Mistakes

- Using comments to state obvious things:
  `x = 5  # x is 5` (Not useful)
- Outdated comments (not matching code):
  `y = 10  # y stores name` (Misleading)
- Too many comments (makes code noisy):
  `a = b + c  # a equals b plus c` (Unnecessary)

Key Takeaways

- Indentation defines block structure in Python.
- Use consistent indentation (prefer 4 spaces).
- Comments make code readable and maintainable.
- Good code = Clear structure + Helpful comments.

Pro Tip: Write code for machines to run, but write it well for humans to read!

Clean indentation + Meaningful comments = Readable, Maintainable, and Professional Python Code!

Reading Input, Printing Output in Python

Programs interact with users through Input and Output.
Python provides simple functions to read input from the user and display output on the screen.

1. READING INPUT

Use the `input()` function to read data from the user.
It always returns data as a string.

Syntax

```
variable = input("Prompt message")
```

How it works

- The **prompt message** is displayed to the user.
- The user types something and presses Enter.
- The typed data is stored as a string in the variable.

Example 1: Simple Input

```
name = input("Enter your name: ")
print("Hello,", name)
```

Run:

```
Enter your name: Alice
Hello, Alice
```

Example 2: Input with Type Conversion

```
age = int(input("Enter your age: "))
height = float(input("Enter your height (in m): "))
print("Age:", age, "years")
print("Height:", height, "meters")
```

Run:

```
Enter your age: 20
Enter your height (in m): 1.72
Age: 20 years
Height: 1.72 meters
```

Important Points

- `input()` returns data as a string. Convert it to `int()`, `float()`, or other types when needed.
- Use meaningful prompts to guide the user.
- The program waits for user input at the `input()` line.

Common Mistakes

- Forgetting type conversion:
  ```
  age = input("Enter age: ") + 5    # Error! (string + int)
  ```
- Using `input()` without a prompt:
  ```
  name = input()                    # No message for user
  ```
- Expecting numbers but not converting:
  ```
  x = input("Enter a number: ") * 2   # Repeats string, not multiply!
  ```

Quick Tips

- Always convert input to the required type.
- Validate user input when necessary.
- Use clear prompts and messages.

2. PRINTING OUTPUT

Use the `print()` function to display output.
It can print text, variables and expressions.

Syntax

```
print(value1, value2, ..., sep=' ', end='\n')
```

How it works

- By default, items are separated by a space (`sep=' '`).
- By default, `print()` moves to a new line after printing (`end='\n'`).
- You can change `sep` and `end` for customized output.

Commonly Used Parameters

Parameter	Default	Description	Example
sep	' ' (space)	Separator between items	`print(1, 2, 3, sep=", ")` Output: 1, 2, 3
end	'\n' (new line)	What to print at the end	`print("Hello", end="! ")` `print("Python")` Output: Hello! Python
file	sys.stdout	Where to print	(Advanced topic)
flush	False	Force immediate output	(Advanced topic)

Examples

1. Printing Text

```
print("Hello, World!")
```

Output:
```
Hello, World!
```

2. Printing Variables and Expressions

```
x = 10
y = 5
print("Sum =", x + y)
```

Output:
```
Sum = 15
```

3. Multiple Items

```
name = "Alice"
age = 20
print("Name:", name, "Age:", age)
```

Output:
```
Name: Alice Age: 20
```

4. Using sep and end

```
print("A", end=" ")
print("B", end=" ")
print("C")
```

Output:
```
A B C
```

Example: Complete Program

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Hello,", name + "!")
print("You will be", age + 1, "next year.")
```

Sample Run

```
Enter your name: Bob
Enter your age: 19
Hello, Bob!
You will be 20 next year.
```

Key Takeaways

- `input()` is used to read data from the user.
- `print()` is used to display output.
- Input helps programs become interactive.
- Clear output makes programs user-friendly.

Input takes data from the user. **Output** shows results to the user.
Together they make Python programs **interactive** and useful!

Pro Tip: Test your program with different inputs to make it robust!

Type Conversions in Python

Type conversion (or type casting) is the process of converting one data type into another. It helps in performing operations between different types of data.

```
x = 10        # int
y = 3.14      # float
s = '25'      # str

a = int(s)    # str to int
b = float(x)  # int to float
c = str(y)    # float to str
```

Key Idea

Convert data to the right type whenever required. Wrong type can cause errors or unexpected results.

1. Why Type Conversion?

- User input is always taken as string.
- Operations between different types may not be allowed.
- Converting to appropriate type avoids errors and ensures correct results.

Example:

```
a = input("Enter a number: ")   # string
b = a + 5          # Error
b = int(a) + 5     # Works
```

```
TypeError: can only concatenate str
(not "int") to str
```

2. Built-in Conversion Functions in Python

Function	Converts To	Example	Result

`int(x)`	Integer	`int('25')`
		`int(3.9)`
		`int(True)`
			25
			3
			1

`float(x)`	Floating point	`float('3.14')`
		`float(10)`
		`float(True)`
			3.14
			10.0
			1.0

`str(x)`	String	`str(123)`
		`str(3.14)`
		`str(True)`
			'123'
			'3.14'
			'True'

`bool(x)`	Boolean	`bool(0)`
		`bool(25)`
		`bool('')`
		`bool('Hello')`
			False
			True
			False
			True

3. Common Conversions with Examples

Conversion	Code	Output	Explanation
String to int	`x = int('42')` `print(x)`	42	Numeric string to integer
String to float	`y = float('3.75')` `print(y)`	3.75	Numeric string to float
Int to float	`z = float(10)` `print(z)`	10.0	Integer to floating point
Float to int	`a = int(7.99)` `print(a)`	7	Decimal part is truncated
Int to string	`b = str(100)` `print(b)`	'100'	Integer to string
Float to string	`c = str(3.5)` `print(c)`	'3.5'	Float to string

Bool conversions	`print(bool(0))`
	`print(bool(1))`
	`print(bool(''))`
	`print(bool('Hi'))`
		False
		True
		False
		True
			Zero or empty → False
			Non-zero or non-empty → True

4. User Input and Type Conversion

`input()` always returns data as string. Convert it to required type.

```
age = int(input("Enter your age: "))    # convert to int
price = float(input("Enter price: "))   # convert to float
name = input("Enter your name: ")       # already string
```

Sample Run

```
Enter your age: 19
Enter price: 99.95
Enter your name: Alice
```

→

Stored As

```
age   → 19      (int)
price → 99.95   (float)
name  → 'Alice' (str)
```

5. Decimal to Integer Conversion

`int()` removes the fractional part (truncates toward zero).

```
print(int(9.7))     # 9
print(int(-9.7))    # -9
```

Note: Use `round()`, `floor()`, `ceil()` for other needs.

Function	Meaning	Example (x = 3.6)	Output
`round(x)`	Rounds to nearest	`round(x)`	4
`math.floor(x)`	Rounds down	`floor(x)`	3
`math.ceil(x)`	Rounds up	`ceil(x)`	4

6. Important Points

- Choose the correct type for your data.
- Always validate and convert user input.
- Invalid conversions raise ValueError.

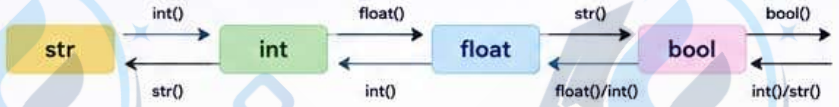
Example of Error

```
x = int('abc')    # invalid literal
# ValueError: invalid literal for int() with base 10: 'abc'
```

Safe Conversion Using try-except

```
try:
    x = int(input("Enter a number: "))
    print("You entered:", x)
except ValueError:
    print("Invalid input! Please enter a valid number.")
```

7. Conversion Summary



Invalid Conversions (Examples)

- `int('12.5')` # ValueError
- `float('abc')` # ValueError
- `int(True.5)` # ValueError

Right type, right result! Use type conversion smartly to write error-free and efficient Python programs.

The type() Function and Is Operator in Python

Python provides built-in tools to inspect data and compare objects.
The type() function tells us the data type of a value, and the is operator checks object identity.

1. The type() Function

The type() function returns the type (class) of the object passed as an argument.

Syntax

```
type(object)
```

How it works

- It returns the class of the object as a type object.
- The output is of type 'type'.

Examples

```
type(10)          # <class 'int'>
type(3.14)        # <class 'float'>
type('Hello')     # <class 'str'>
type([1, 2, 3])   # <class 'list'>
type((1, 2))      # <class 'tuple'>
type(True)        # <class 'bool'>
type({'a':1})     # <class 'dict'>
type(None)        # <class 'NoneType'>
```

Common Return Types

Data Type	Example	Data Type	Example
int	type(5) → <class 'int'>	tuple	type((1,2)) → <class 'tuple'>
float	type(3.5) → <class 'float'>	dict	type({'a':1}) → <class 'dict'>
str	type('hi') → <class 'str'>	bool	type(False) → <class 'bool'>
list	type([1,2]) → <class 'list'>	NoneType	type(None) → <class 'NoneType'>

Practical Use

- To check the type of data stored in a variable.
- To write type-safe programs.
- To perform different actions based on data type.

Example

```
x = input("Enter something: ")

print(type(x))          # Always <class 'str'>
n = int(x)              # Convert to int
print(type(n))          # Now <class 'int'>
```

Keep in Mind

- type(1) and type(True) are different: <class 'int'> vs <class 'bool'>
- None has its own type: <class 'NoneType'>

2. The is Operator

The is operator checks whether two variables point to the same object in memory (identity), not just equal values.

Syntax

```
object1 is object2
```

How it works

- Returns True if both refer to the same object.
- Returns False if they refer to different objects, even if values are equal.

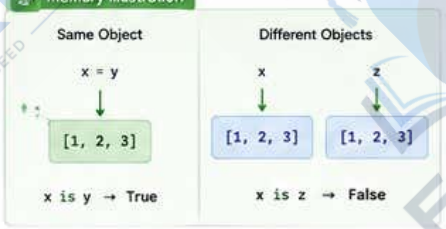
Examples

```
a = 10
b = 10
print(a is b)           # True (same object)

x = [1, 2, 3]
y = x
z = [1, 2, 3]
print(x is y)           # True  (same object)
print(x is z)           # False (different objects)
```

is vs == (Important Difference)

Operator	What it checks	Example	Output
==	Value Equality (values are equal)	a = [1,2] b = [1,2] print(a == b)	True
is	Identity Equality (same object in memory)	a = [1,2] b = [1,2] print(a is b)	False

Memory Illustration



Common Use Cases

- Check if a variable is None: if value is None:
- Check for singletons like True, False, None.
- Useful with immutable singletons.

Examples

```
user_input = None
if user_input is None:
    print("No input provided")

if flag is True:
    print("Flag is True")
```

Common Mistakes

- Using is to compare values → may give unexpected results.
- Do not use 'is' with numbers, strings, or lists for value comparison. Use '==' instead.

Wrong ✖

```
a = 1000
b = 1000
print(a is b)  # May be False
```

Right ✔

```
a = 1000
b = 1000
print(a == b)  # True
```

Key Takeaway: Use type() to know what a value is.
Use is to check whether two variables are the exact same object.

Pro Tip: When in doubt:
Use '==' for value comparison, and 'is' for identity comparison.

Dynamic and Strongly Typed Language in Python

Python is a **Dynamic** and **Strongly Typed** language.
This combination gives flexibility to the programmer while ensuring safety and reliability in code.

1. DYNAMIC TYPING

In Python, the type of a variable is determined at runtime, not at compile time.

Key Points

- You don't need to declare the type of a variable.
- A variable can hold different types of values at different times.
- Python automatically manages memory and type changes during execution.

Examples

```
x = 10              # int
print(type(x))      # <class 'int'>

x = "Hello")        # str
print(type(x))      # <class 'str'>

x = 3.14            # float
print(type(x))      # <class 'float'>

x = [1, 2, 3]       # list
print(type(x))      # <class 'list'>
```

Another Example

```
age = 25
name = "Alice"
age = "twenty five"     # same variable, new type
is_active = True
is_active = 0           # now it becomes int
```

2. STRONGLY TYPED

Python does not allow operations between incompatible types.

Key Points

- Every value has a specific type.
- Python prevents mixing incompatible types in operations.
- Helps in catching errors and prevents unintended behavior.

Examples

Valid Operations (Same or Compatible Types)

```
a = 10
b = 20
print(a + b)          # 30

c = 3.5
d = 2.5
print(c + d)          # 6.0

s1 = "Hello"
s2 = "Python"
print(s1 + " " + s2)  # Hello Python
```

Invalid Operations (Different Types)

```
a = 10
b = "20"
print(a + b)
# TypeError: unsupported
# operand type(s) for +:
# 'int' and 'str'

x = 5
y = [1, 2, 3]
print(x + y)
# TypeError: unsupported
# operand type(s) for +:
# 'int' and 'list'
```

Why is this combination powerful?

- Dynamic Typing → More flexibility, less code.
- Strong Typing → Fewer bugs, safer programs.
- Ideal balance of programmer productivity and program reliability.

Dynamic vs Strongly Typed

Feature	Dynamic Typing	Strongly Typed
Type Checking Time	At Runtime	During Operation
Variable Type	Can change during program execution	Type is fixed for a value
Declaration	No need to declare type	Type is inherent to value
Allows Mixing Types	Yes (variable can hold any type)	No (operations on incompatible types are not allowed)
Example	`x = 10; x = "Hello"; x = 3.14`	`10 + '20'` ✗, `10 + 20` ✓

Real-World Analogy

- Dynamic Typing is like a box that can hold anything at different times.
- Strong Typing is like rules that allow only compatible items to be combined or used together.

Common Mistakes

- Thinking Python ignores types completely.
  → It checks types during operations.
- Expecting automatic conversions.
  → Convert explicitly using type conversion functions.
- Mixing types in operations.
  → Leads to TypeError.

Type Conversion is Explicit

```
a = 10
b = "20"
c = a + int(b)      # Works: 30
d = str(a) + b      # Works: "1020"
```

Check Types Anytime

```
x = [1, 2, 3]
print(type(x))                  # <class 'list'>
print(isistance(x, list))       # True
print(isistance(x, str))        # False
```

`isistance()` is useful to check type or class.

Key Takeaways

- Python is Dynamic → types decided at runtime.
- Python is Strongly Typed → no invalid operations between incompatible types.
- Makes Python easy to write and safe to run!

Dynamic gives you freedom, **Strong typing** gives you safety.
That's what makes Python **simple, powerful** and **reliable**!

Pro Tip: When in doubt, check the type!
Use `type()` and `isistance()` to write better Python code.


1010101010
0101010101
1010101010
0101010101
Unit - IV
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Control Flow Statements: The if, if-else

Control Flow Statements allow a program to make **decisions** and execute different blocks of code based on **conditions**.

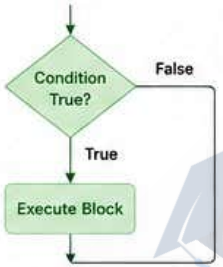
1. The if Statement

The if statement is used to execute a block of code only if a condition is True.

Syntax

```
if condition:
    # block of code
```

Flowchart



Example 1: Check if a number is positive

```
num = 8
if num > 0:
    print("Number is positive")
```

Output

```
Number is positive
```

Example 2: Check if a user is eligible to vote

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

Output

```
You are eligible to vote.
```

Important Notes

- The condition after if must be True or False.
- If the condition is True, the indented block runs.
- If False, the block is skipped.
- Indentation (usually 4 spaces) is mandatory in Python.

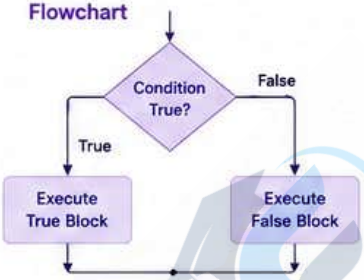
2. The if-else Statement

The if-else statement is used to execute one block of code if the condition is True, and another block if it is False.

Syntax

```
if condition:
    # block of code if True
else:
    # block of code if False
```

Flowchart



Example 1: Check if a number is positive or non-positive

```
num = -5
if num > 0:
    print("Positive")
else:
    print("Non-positive")
```

Output

```
Non-positive
```

Example 2: Find bigger of two numbers

```
a = 12
b = 20
if a > b:
    print("a is bigger")
else:
    print("b is bigger")
```

Output

```
b is bigger
```

3. How They Work

- Python evaluates the condition.
- If the condition is True, the if block executes.
- If the condition is False:
  - In if: nothing happens (block skipped).
  - In if-else: the else block executes.

if vs if-else

Feature	if Statement	if-else Statement
Purpose	Executes code only if condition is True.	Executes code if True, otherwise executes another block.
Else Part	No	Yes
Use When	When only one condition needs to be handled.	When two possibilities need to be handled.
Example Use	Check if a number is positive.	Check if a number is positive or negative.

Common Mistakes

- Forgetting the colon (:) after if or else.
- Wrong indentation (leads to IndentationError).
- Using = instead of == in conditions.
- Expecting else to run when if condition is True.

More Examples

Example: Check even or odd

```
n = 7
if n % 2 == 0:
    print("Even")
else:
    print("Odd")
```

Example: Grade based on marks

```
marks = 85
if marks >= 40:
    print("Pass")
else:
    print("Fail")
```

Quick Rules

- Use if for one-way decisions.
- Use if-else for two-way decisions.
- Conditions can use comparison, logical, and membership operators.
- Keep your code clean and properly indented.

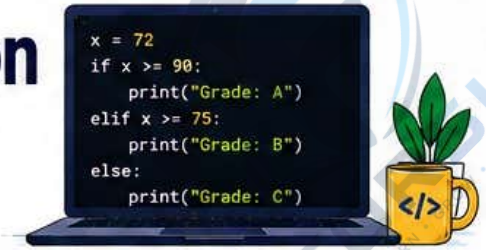
Key Takeaway

The if statement lets your program act when a condition is True.

The if-else statement lets your program choose between two paths.

Decisions make programs smart!
Use if and if-else to control the flow.

if-elif-else Decision Control Statement in Python



The `if-elif-else` statement is used to make decisions among multiple conditions.
Python checks the conditions from top to bottom and executes the first True block.

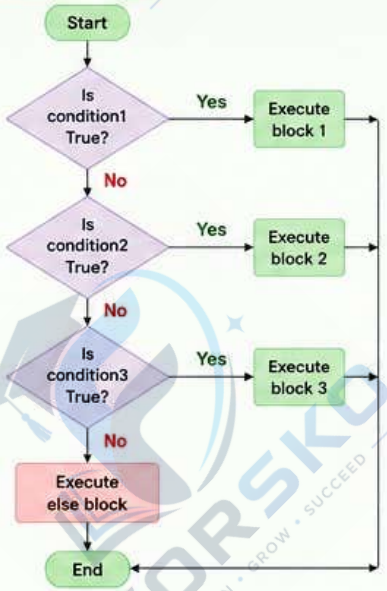
1. Syntax

```
if condition1:
    # block of code if condition1 is True
elif condition2:
    # block of code if condition2 is True
elif condition3:
    # block of code if condition3 is True
    ......
else:
    # block of code if all conditions are False
```

Key Points

- The conditions are checked in order.
- As soon as one condition is True, its block executes and the rest are skipped.
- else is optional, but it is executed if none of the conditions are True.
- Indentation is mandatory.

2. Flowchart



3. Example Program: Grade Finder

```
marks = int(input("Enter your marks: "))
if marks >= 90:
    print("Grade: A (Excellent)")
elif marks >= 75:
    print("Grade: B (Good)")
elif marks >= 50:
    print("Grade: C (Average)")
else:
    print("Grade: F (Fail)")
```

Sample Runs

Input (marks)	Output
95	Grade: A (Excellent)
80	Grade: B (Good)
60	Grade: C (Average)
30	Grade: F (Fail)

4. How it Works

1. Python checks the first condition (marks >= 90).
2. If True → executes that block and stops.
3. If False → moves to the next condition (marks >= 75).
4. This continues until one condition is True.
5. If none are True → else block executes.

5. Truth Conditions (Example)

marks >= 90	marks >= 75	marks >= 50	Executed Block
True	–	–	if block
False	True	–	elif (>=75) block
False	False	True	elif (>=50) block
False	False	False	else block

6. More Examples

A. Find Largest of 3 Numbers

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))

if a >= b and a >= c:
    print("a is largest")
elif b >= c:
    print("b is largest")
else:
    print("c is largest")
```

B. Day Name from Number

```
day = int(input("Enter day (1-7): "))
if day == 1:
    print("Monday")
elif day == 2:
    print("Tuesday")
elif day == 3:
    print("Wednesday")
elif day == 4:
    print("Thursday")
elif day == 5:
    print("Friday")
elif day == 6:
    print("Saturday")
elif day == 7:
    print("Sunday")
else:
    print("Invalid day number")
```

C. Discount Calculator

```
amount = float(input("Enter amount: "))
if amount >= 5000:
    discount = 0.20
elif amount >= 2000:
    discount = 0.10
elif amount >= 1000:
    discount = 0.05
else:
    discount = 0.0

payable = amount - (amount * discount)
print("Payable amount:", payable)
```

7. Common Mistakes

- Forgetting colons (:) after if, elif, else.
- Wrong indentation → IndentationError.
- Using = instead of == for comparison.
- Placing else before elif.
- Expecting multiple conditions to run. (Only the first True block runs.)

Wrong Example

```
if x > 10
    print("x is greater")
elif x > 5
    print("x is greater than 5")
else
    print("Small number")
```

- Missing colons
- Wrong indentation

8. Quick Takeaway

- Use if for single condition.
- Use if-elif-else for multiple conditions.
- Conditions are checked from top to bottom.
- First True block executes.
- else runs only when all conditions are False.

Pro Tips

- Keep conditions simple and in logical order.
- Use meaningful condition expressions.
- Use elif instead of multiple if to avoid unnecessary checks.

Think → Check → Decide → Execute

Use `if-elif-else` to make your programs smart and efficient!

Practice more, logic more, bug less!

Nested if Statement in Python

```
x = 15
if x > 0:
    if x % 2 == 0:
        print("Positive Even")
    else:
        print("Positive Odd")
else:
    print("Non-positive")
```

A nested if statement is an if statement inside another if, elif or else block.
It is used to make more complex decisions by checking multiple conditions.

1. What is Nested if?

When a condition is True, Python moves inside the block and checks the next condition.

Key Points

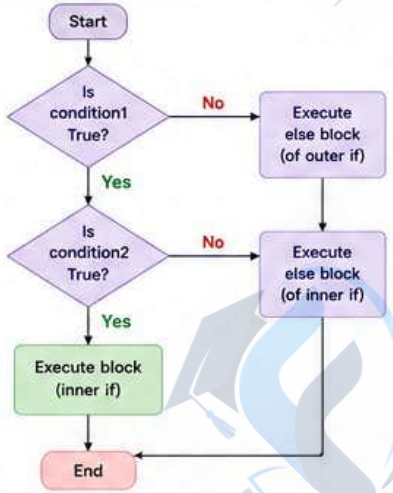
- The inner if is executed only when the outer if (or elif) condition is True.
- It helps to test multiple conditions step by step.
- You can have multiple levels of nesting.
- Proper indentation is very important.

2. Syntax and Structure

```
if condition1:
    if condition2:
        # block of code if both condition1 and condition2 are True
    else:
        # block of code if condition2 is False
else:
    # block of code if condition1 is False
```

You can also use elif with nesting.
The inner block can have if, elif, else again.

3. Flowchart



4. Example Program with Explanation

```
1  num = int(input("Enter a number: "))
2
3  if num > 0:                          # Outer if
4      if num % 2 == 0:                 # Inner if
5          print("Positive Even Number")
6      else:                            # Inner else
7          print("Positive Odd Number")
8  else:                                # Outer else
9      print("Non-positive Number")
```

Sample Runs

Input (num)	Output	Explanation
10	Positive Even Number	Outer if True, Inner if True
7	Positive Odd Number	Outer if True, Inner if False
-3	Non-positive Number	Outer if False

5. More Examples

A. Check Grade Based on Marks

```
marks = int(input("Enter marks (0-100): "))
if marks >= 60:
    if marks >= 90:
        print("Grade: A")
    elif marks >= 75:
        print("Grade: B")
    else:
        print("Grade: C")
else:
    print("Grade: F (Fail)")
```

First checks if passed (>=60).
If passed, then checks for A or B, otherwise C. If not passed, then F.

B. Check Eligibility for Driving

```
age = int(input("Enter age: "))
has_license = input("Do you have license? (y/n): ")
if age >= 18:
    if has_license.lower() == 'y':
        print("You can drive.")
    else:
        print("You need a license.")
else:
    print("You must be 18 or older.")
```

Outer if checks age.
If age condition True, inner if checks license.

C. Largest of Three Numbers

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))
if a >= b:
    if a >= c:
        print("Largest is", a)
    else:
        print("Largest is", c)
else:
    if b >= c:
        print("Largest is", b)
    else:
        print("Largest is", c)
```

Compare a and b first, then compare with c.

6. Common Mistakes

- Incorrect indentation
  Leads to IndentationError.
- Forgetting colon (:) after if or else.
- Placing else without matching if.
  Else must belong to the naerest if.
- Over-complicating logic.
  Keep conditions simple and readable.

Wrong Example

```
if x > 0                    ⟷ Missing colon
    if x % 2 == 0:
        print("Even")
    else:                   ⟷ Wrong indentation
        print("Odd")
```

Correct Example

```
if x > 0:
    if x % 2 == 0:
        print("Even")
    else:
        print("Odd")
```

7. When to Use Nested if

- When a condition depends on another condition being True.
- When decisions have multiple levels.
- When you need more precise control over the flow of execution.

Quick Takeaway

- Nested if = decision within a decision.
- Check conditions from outer to inner.
- Only True paths go deeper.
- Good indentation = bug free code!

Break complex problems into smaller decisions using Nested if.

Think step by step. Code smart!

The while Loop in Python

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

```
# Output
1
2
3
4
5
```

The while loop is used to execute a block of code repeatedly as long as the specified condition is True.

1. What is a while Loop?

The while loop repeats a block of code as long as the condition is True. When the condition becomes False, the loop stops.

Key Points

- Condition is checked before each iteration.
- If condition is False initially, loop body will not execute.
- Used when the number of iterations is unknown.
- Proper update of the condition is essential to avoid infinite loops.

2. Syntax

```
while condition:
    # block of code
    # update condition
```

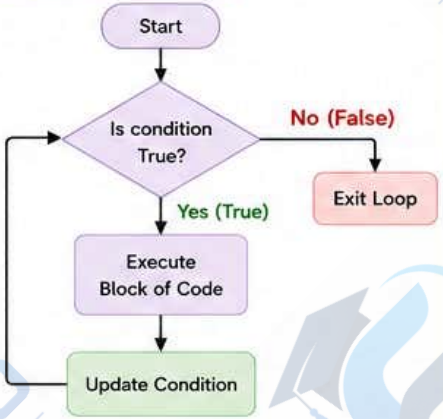
Example Syntax

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Equivalent to:

Repeat the block of code while condition remains True.

3. How it Works?



4. Example: Print 1 to 5

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Output

```
1
2
3
4
5
```

Step-by-step Execution

Iteration	count (before)	Condition (count <= 5)	Action	count (after)
1	1	True	print 1	2
2	2	True	print 2	3
3	3	True	print 3	4
4	4	True	print 4	5
5	5	True	print 5	6
Next	6	False	Exit loop	--

5. More Examples

A. Sum of first n natural numbers

```
n = int(input("Enter n: "))
i = 1
sum = 0
while i <= n:
    sum += i
    i += 1
print("Sum =", sum)
```

Example:
Enter n: 5
Sum = 15

B. Factorial of a number

```
n = int(input("Enter n: "))
fact = 1
i = 1
while i <= n:
    fact *= i
    i += 1
print("Factorial =", fact)
```

Example:
Enter n: 5
Factorial = 120

C. Count digits in a number

```
num = int(input("Enter a number: "))
count = 0
while num > 0:
    num //= 10
    count += 1
print("Number of digits =", count)
```

Example:
Enter a number: 12345
Number of digits = 5

6. Common Mistakes

- **Infinite Loop:** Forgetting to update the condition inside the loop.

```
i = 1
while i <= 5:
    print(i)      # i is never updated
```

- **Wrong Condition:** Using wrong operators.

```
while i = 5:   # = is assignment, not comparison
```

- **Indentation Error:** Incorrect indentation of loop body.

```
while i <= 5:
print(i)
```

```
while i <= 5:
    print(i)
```

7. Useful Variations

- `while True:`
  `# Infinite loop (use break to stop)`
- Using `break`:

```
i = 1
while i <= 10:
    if i == 7:
        break        # exits loop when i == 7
    print(i)
    i += 1
```

- Using `continue`:

```
i = 1
while i <= 5:
    i += 1
    if i == 3:
        continue     # skip printing 3
    print(i)
```

8. When to Use while Loop?

- When the number of repetitions is not known in advance.
- When we need to keep repeating until a condition changes.
- When reading input until a specific condition is met.

Key Takeaway

while loop keeps running as long as the condition is True.

Update the condition → Avoid infinite loop → Write correct logic.

Pro Tip

Always test your loop with small inputs first and make sure your condition will eventually become False.

The for Loop in Python

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

```
# Output
1
2
3
4
5
```

The for loop is used to iterate (repeat) over a sequence (such as list, tuple, string, range, etc.) or other iterable objects.

1. What is a for Loop?

The for loop is used to execute a block of code for each item in a sequence (or for a fixed number of times using `range()`).

Key Points

- It is used when the number of iterations is known.
- It automatically moves to the next item after each iteration.
- No need to update a counter manually.
- Cleaner and easier to read than `while` loops in many cases.

2. Syntax

```
for variable in sequence:
    # block of code
```

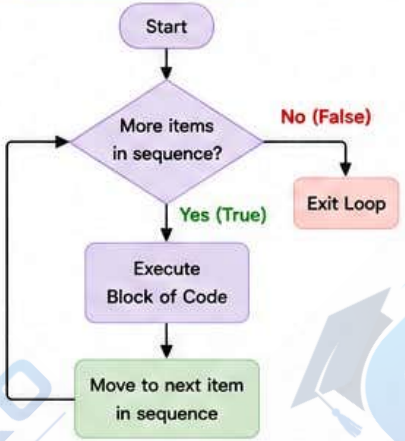
Explanation

- **'variable'** takes the value of each item one by one.
- **'sequence'** can be any iterable object.

Example Syntax

```
for x in [10, 20, 30]:
    print(x)
```

3. How it Works?



4. Example Program: Print items of a list

```
fruits = ["apple", "banana", "mango"]
for fruit in fruits:
    print(fruit)
```

Output

```
apple
banana
mango
```

Step-by-step Execution

Iteration	fruit (current item)	Action	Output so far
1	apple	print(apple)	apple
2	banana	print(banana)	apple banana
3	mango	print(mango)	apple banana mango
Next	–	No more items → Exit loop	–

5. More Examples

A. Using range()

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

Output:

```
1
2
3
4
5
```

B. Sum of first n natural numbers

```
n = int(input("Enter n: "))
s = 0
for i in range(1, n+1):
    s += i
print("Sum =", s)
```

Example (n = 5)
Output: Sum = 15

C. Iterate over a string

```
for ch in "Python":
    print(ch, end=' ')
```

Output:

```
P y t h o n
```

D. Iterate over a tuple

```
t = (10, 20, 30)
for x in t:
    print(x)
```

Output:

```
10
20
30
```

6. Common Mistakes

- Forgetting the colon (`:`) after `for` statement.

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

- Wrong indentation.

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

- Using `range()` incorrectly.

```
for i in range(5, 1):  # won't run (step is positive)
    print(i)
```

- Modifying the loop variable inside the loop.

```
for i in range(5):
    i = i + 1    # has no effect on next iteration
```

7. Useful Variations

Using range(start, stop, step)

```
for i in range(2, 11, 2):   # 2 to 10, step 2
    print(i)
```

Output: 2 4 6 8 10

Using enumerate() (index + value)

```
names = ["Ali", "Sara", "John"]
for idx, name in enumerate(names):
    print(idx, name)
```

Output:

```
0 Ali
1 Sara
2 John
```

Loop with else

```
for i in range(3):
    print(i)
else:
    print("Loop finished successfully")
```

Output:

```
0
1
2
Loop finished successfully
```

8. When to Use for Loop?

- When you know how many times to repeat.
- When you are iterating over a sequence (list, tuple, string, set, dict, etc.).
- When you need cleaner and more readable code.

Key Takeaway

The for loop makes your code short, clean and efficient when the number of iterations is known.
Use it to iterate, process and automate tasks easily!

Pro Tip

- Use `range()` for numbers.
- Use `for` with collections to access each item.
- Prefer `for` over `while` when the count is known.

The continue and break Statements in Python

Both **continue** and **break** are used to control the flow of loops (**for** and **while**). They help us skip iterations or terminate the loop before it completes.

1. continue Statement

The **continue** statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

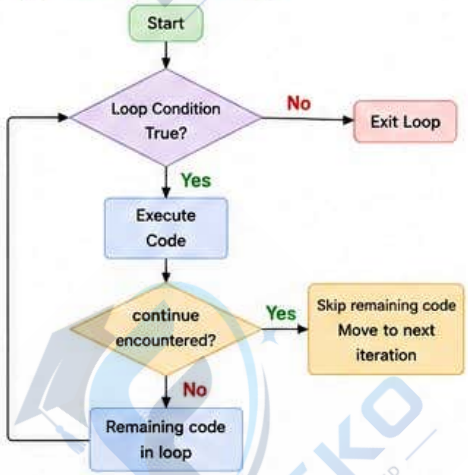
Syntax

```
continue
# used inside a loop
```

How it works?

- Rest of the code in the loop is skipped.
- Control moves to the next iteration.
- Loop does not terminate.

Flowchart (continue)



Examples with continue

A. Skip even numbers and print odd numbers

```
for i in range(1, 8):
    if i % 2 == 0:
        continue    # skip even numbers
    print(i)
```

Output:
```
1
3
5
7
```

Here, when `i` is even, `continue` skips `print(i)` and moves to the next number.

B. Skip a particular value

```
for name in ["Ali", "Sara", "John", "Eva"]:
    if name == "Sara":
        continue    # skip Sara
    print(name)
```

Output:
```
Ali
John
Eva
```

When `name` is `"Sara"`, it is skipped and loop `continue` with the next item.

2. break Statement

The **break** statement terminates the loop completely and transfers the control to the statement immediately after the loop.

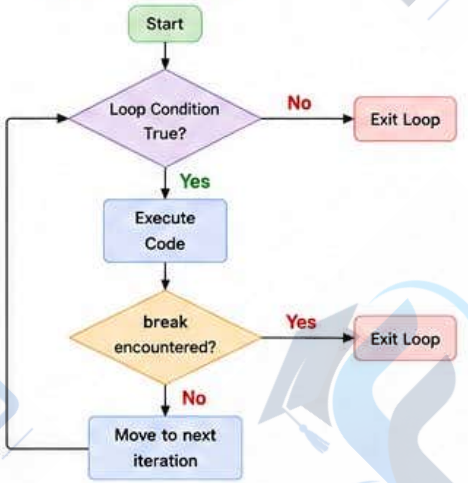
Syntax

```
break
# used inside a loop
```

How it works?

- When **break** is encountered, the loop stops.
- No more iterations are executed.
- Control moves to the next statement after the loop.

Flowchart (break)



Examples with break

A. Stop the loop when a condition is met

```
for i in range(1, 10):
    if i == 5:
        break    # stop when i is 5
    print(i)
```

Output:
```
1
2
3
4
```

When `i` becomes 5, **break** is executed and loop terminates.

B. Search and stop when item is found

```
numbers = [4, 7, 2, 9, 5]
for x in numbers:
    if x == 9:
        print("Found 9")
        break
    print(x)
```

Output:
```
4
7
2
Found 9
```

When 9 is found, **break** stops the loop.

3. continue vs break

Feature	continue	break
Purpose	Skips current iteration	Terminates the loop
Effect	Moves to next iteration	Exits the loop completely
Loop	Loop continues	Loop stops
Use When	You want to skip some iterations	You want to stop the loop based on a condition

4. Common Mistakes

- Forgetting indentation after **continue** or **break**.
- Using **continue** or **break** outside a loop (leads to SyntaxError).
- Expecting **break** to skip only current iteration (it stops the loop).
- Forgetting that code after **continue** will not run in that iteration.

Wrong Example

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

`print(i)` is never executed. (It is after **continue**.)

Correct Example

```
for i in range(5):
    continue
    # next iteration starts
```

5. Key Takeaways

- Use **continue** to skip unwanted iterations.
- Use **break** to terminate the loop early.
- Both are very useful for writing efficient and clean code.
- They work with both **for** and **while** loops.

Pro Tip

Use continue to avoid deep nesting of if conditions.
Use break when the goal is achieved and no more searching is needed.

Remember: Loops are powerful, but controlling them correctly with **continue** and **break** makes your code smarter and more effective!

Functions: Built-In Functions in Python

```
print(len("Python"))      # 6
print(max([4, 1, 7, 2]))  # 7
print(type(10.5))         # <class 'float'>
```

Built-in functions are pre-defined in Python and are always available for use.
They perform common tasks, making our programs shorter, easier and more efficient.

1. What are Built-In Functions?

These functions are part of Python's standard library. You can use them directly without importing any module.

Example: `print()`, `len()`, `type()`, `max()`, `min()`, `sum()`, `input()` etc.

2. Categories of Built-In Functions

- Input / Output Functions
- Type Conversion Functions
- Numeric Functions
- Sequence (String / List / Tuple) Functions
- Set / Dictionary Functions
- Utility Functions
- Others

4. Commonly Used Built-In Functions with Examples

Category	Function	Description	Example	Output
Input / Output Functions	print()	Displays the given value(s) on the screen.	`print("Hello Python")`	Hello Python
	input()	Takes input from the user as a string.	`name = input("Enter name: ")`	(user input)
Type Conversion Functions	int()	Converts value to integer.	`int("25")`	25
	float()	Converts value to float.	`float("3.14")`	3.14
	str()	Converts value to string.	`str(100)`	'100'
	type()	Returns the type of an object.	`type(10.5)`	<class 'float'>
Numrce Functions	abs()	Returns absolute value.	`abs(-9)`	9
	pow(x, y)	Returns x raised to the power y.	`pow(2, 3)`	8
	round(x, n)	Rounds x to n digits.	`round(3.14159, 2)`	3.14
	max(iterable)	Returns largest item.	`max([4, 1, 7, 2])`	7
	min(iterable)	Returns smallest item.	`min("python")`	'h'
	sum(iterable)	Returns sum of items.	`sum([1, 2, 3, 4])`	10
Sequence (String / List / Tuple) Functions	len(s)	Returns length (number of items).	`len([10, 20, 30])`	3
	sorted(iterable)	Returns a sorted list.	`sorted([3, 1, 2])`	[1, 2, 3]
	reversed(seq)	Returns reverse iterator.	`list(reversed("abc"))`	['c', 'b', 'a']
	enumerate(seq)	Returns index and value as pairs.	`list(enumerate(['a','b"]))`	[(0, 'a'), (1, 'b')]
Set / Dictionary Functions	len(set/dict)	Returns number of items.	`len({'a':1, 'b':2})`	2
	keys()	Returns all keys of dict.	`{'a':1,'b':2}.keys()`	dict_keys(['a', 'b'])
	values()	Returns all values of dict.	`{'a':1,'b':2}.values()`	dict_values([1, 2])
	items()	Returns key-value pairs.	`{'a':1}.items()`	dict_items([['a', 1]])
Utility Functions	all(iterable)	Returns True if all items are True.	`all([True, 1, 'yes"])`	True
	any(iterable)	Returns True if any item is True.	`any([0, False, 3])`	True
Others	id(obj)	Returns identity of object (unique address).	`id(10)`	(some integer)
	help(obj)	Shows help/documentation for an object.	`help(len)`	(help text)
	dir(obj)	Returns list of attributes and methods of object.	`dir([])`	(list of attributes)

. Examples in Action

```
# Length and type
text = "Python"
print(len(text))
print(type(text))
```

Output:
```
6
<class 'str'>
```

```
# Numeric functions
nums = [5, 9, 2, 8]
print(max(nums))
print(min(nums))
print(sum(nums))
```

Output:
```
9
2
24
```

```
# Type conversion
a = int("42")
b = float("3.5")
c = str(100)
print(a, b, c)
```

Output:
```
42 3.5 100
```

```
# Sequence functions
lst = [10, 20, 30]
print(sorted(lst))
print(reversed(lst))
```

Output:
```
[10, 20, 30]
<list_reverseiterator
  object at 0x...>
```

```
# Dictionary functions
d = {'x': 1, 'y': 2}
print(d.keys())
print(d.values())
```

Output:
```
dict_keys(['x', 'y'])
dict_values([1, 2])
```

. Notes

- Function names are case-sensitive.
- Some functions work on specific data types.
- Use `help(function_name)` for details.
  Example: `help(len)`
- All built-in functions can be used without importing any module.

. Common Mistakes

- Forgetting parentheses: `print` instead of `print()`
- Using wrong type in function: `max(10)` (max needs iterable)
- Expecting original list to change with `sorted()`
  (It returns a new sorted list)
- Confusing type conversion: `int("3.5")` ✗ (use `float` for decimals)

7. Key Takeaways

- Built-in functions save time and effort.
- They are easy to use and very powerful.
- Know the right function for the right task!

Pro Tip: Use `help(function_name)` to learn more about any built-in function. **Example:** `help(sum)`

"Smart use of built-in functions makes your code Pythonic and efficient!"

Commonly Used Modules in Python

```
# Using a module in Python
import math
print(math.sqrt(16))   # 4.0
print(math.pi)         # 3.141592653589793
```

Modules are Python files containing functions and variables.
They extend Python's functionality and help us write clean, reusable and efficient code.

1. What is a Module?

- A module is a Python file with `.py` extension that contains definitions (functions, classes, variables, etc.).
- We can import a module to use its functionality.
- Example:

```
import math       # import the math module
print(math.sqrt(25))    # 5.0
```

2. How to Use Modules?

1. Using the entire module
```
import module_name
```
2. Importing specific functions/objects
```
from module import name1, name2
```
3. Importing with an alias
```
import module_name as alias
```
4. Importing everything (not recommended)
```
from module import *
```

3. List of Commonly Used Modules

Module	Purpose
math	Mathematical functions
random	Generate random numbers
datetime	Work with dates and times
time	Time related functions
os	Operating system interface
sys	System-specific parameters
re	Regular expressions (pattern matching)
json	Work with JSON data
collections	Specialized container datatypes
statistics	Statistical operations

4. Top Modules with Examples

Module	What it Does	Commonly Used Functions / Attributes	Example	Output

math	Provides mathematical functions and constants.	`sqrt()`, `pow()`, `factorial()`, `sin()`, `cos()`, `log()`, `pi`, `e`	`import math`
			`print(math.sqrt(16))`
			`print(math.pi)`
				`4.0`
				`3.141592653589793`

random	Generates random numbers and performs random operations.	`random()`, `randint()`, `choice()`, `shuffle()`	`import random`
			`print(random.randint(1, 10))`
			`print(random.choice(['a','b','c']))`
				`7` (varies)
				`'b'`

datetime	Manipulates dates and times.	`datetime.now()`, `date.today()`, `timedelta()`	`from datetime import datetime` `print(datetime.now())`	`2024-06-01 12:30:45.123456` (varies)
time	Provides time related functions.	`time()`, `sleep()`, `ctime()`	`import time` `print(time.time())`	`1717234567.85` (epoch time)
os	Interacts with the operating system.	`listdir()`, `mkdir()`, `remove()`, `path.exists()`	`import os` `print(os.listdir())`	`['file1.py', 'notes.txt', 'data']` (varies)
sys	Provides access to system specific parameters.	`argv`, `version`, `exit()`, `path`	`import sys` `print(sys.version)`	`3.12.2 (main, Feb 6 2024, 10:00:00) [GCC ...]`

re	Supports regular expression operations.	`search()`, `match()`, `findall()`, `sub()`	`import re`
			`text = "My number is 123"`
			`print(re.findall(r'\d+', text))`
				`['123']`

json	Encodes and decodes JSON data.	`dumps()`, `loads()`	`import json`
			`data = {'name': 'Aman', 'age': 20}`
			`print(json.dumps(data))`
				`{"name": "Aman", "age": 20}`

collections	Provides specialized container datatypes.	`Counter`, `defaultdict`, `namedtuple`, `deque`	`from collections import Counter`
			`c = Counter('banana')`
			`print(c)`
				`Counter({'a': 3, 'n': 2, 'b': 1})`

statistics	Performs statistical calculations.	`mean()`, `median()`, `mode()`, `stdev()`	`import statistics`
			`nums = [2, 4, 4, 4, 5, 5, 7]`
			`print(statistics.mean(nums))`
				`4.428571428571429`

5. Example Programs

Program: Roll a dice (random module)

```
import random
print(random.randint(1, 6))
```

Output:
```
4
(varies)
```

Program: Current date and time (datetime module)

```
from datetime import datetime
now = datetime.now()
print(now.strftime("%d-%m-%Y %H:%M:%S"))
```

Output:
```
01-06-2024
12:45:30
(varies)
```

Program: List files in current directory (os module)

```
import os
files = os.listdir('.')
print(files)
```

Output:
```
['a.py', 'b.txt',
'images', ...]
(varies)
```

Program: Find all email IDs in a text (re module)

```
import re
text = "Contact us at test@gmail.com
   or info@site.com"
emails = re.findall(r'[\w.]+@[\w.]+', text)
print(emails)
```

Output:
```
['test@gmail.com',
'info@site.com']
```

6. Common Mistakes

- Forgetting to import the module. — `print(math.sqrt(4))  # Error: name 'math' is not defined`
- Using * import (pollutes namespace). — `from math import *  # Not recommended`
- Wrong module name (case sensitive). — `import Math  # Error (should be 'math')`
- Expecting a module to be available without installing (external modules). — Install using `pip install module_name`

7. Key Takeaways

- Modules extend Python's power and reduce our effort.
- Import only what you need.
- Use the right module for the right task.
- Built-in modules are ready to use (no installation needed).
- Explore `help(module_name)` to learn more.

8. Helpful Tips

- Use `dir(module_name)` to see all functions and attributes.
- Use `help(module_name.function)` for detailed help.
- Read official docs: https://docs.python.org/3/library/

Modules are Python's superpowers! Learn, use and explore them to become a better programmer.

Function Definition and Calling the Function

```python
def greet(name):
    print(f"Hello, {name}!")

greet("Python") # Function call
greet("Developer")
```

A function is a block of organized, reusable code that performs a specific task.
We first define (create) the function and then call (use) it whenever needed.

1. What is a Function?

- A function is a named block of code that performs a specific task.
- It improves code reusability, readability and reduces repetition.
- Functions are executed only when they are called.

2. Why Use Functions?

- Avoid repeating the same code.
- Make programs easier to read and maintain.
- Break large programs into smaller parts.
- Promotes code reusability.

3. Anatomy of a Function

```
def function_name(parameters):      ← Header
    """Docstring (optional)"""      ← Docstring
    # statements (function body)    ← Body
    return value  # (optional)      ← Return
```

4. Function Definition (Syntax)

```
def function_name(parameters):
    """Docstring (optional)"""
    # statements
    return value  # (optional)
```

Explanation

- **def** – keyword to define a function
- **function_name** – any valid name
- **parameters** – inputs to the function (optional)
- **return** – sends a value back (optional)

5. Calling the Function

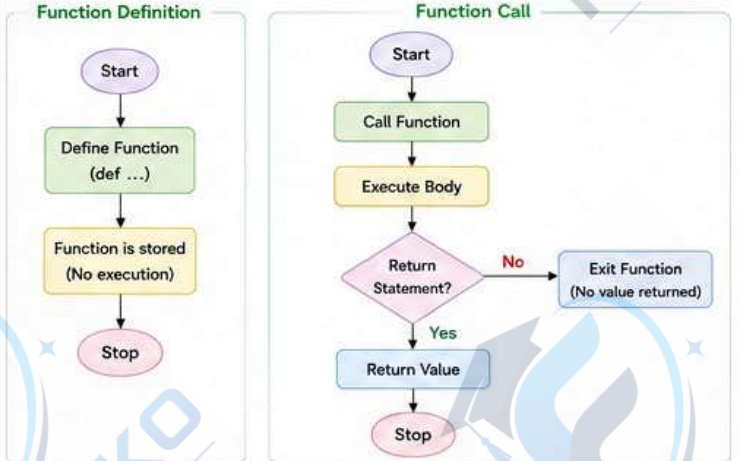
```
result = function_name(arguments)
```

- **function_name** – name of the function
- **arguments** – actual values passed to the function
- **result** – (optional) variable to store returned value

Explanation

- When a function is called, its body executes.
- If it has a **return** statement, the value is returned to the caller.
- If **return** is not used, the function performs tasks without returning value.

6. Flow of Execution



7. Common Mistakes

- Forgetting the colon (:) after def.
- Indentation errors.
- Calling the function before it is defined.
- Mismatching number of arguments.
- Forgetting return when a value is expected.

```
# Error Example
def add(a, b)    # Missing colon
    return a + b
```

8. Examples

1. Function with No Parameters

```
def say_hello():
    print("Hello, World!")

Call:
say_hello()

Output:
Hello, World!
```

2. Function with Parameters

```
def greet(name):
    print("Hello, " + name + "!")

Call:
greet("Alice")

Output:
Hello, Alice!
```

3. Function with Multiple Parameters

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

Call:
result = add(5, 7)
print(result)

Output:
12
```

4. Function with Return Value

```
def square(n):
    return n * n

Call:
s = square(6)
print(s)

Output:
36
```

5. Function Without Return

```
def show(name):
    print("Welcome,", name)

Call:
show("Bob")

Output:
Welcome, Bob
```

9. More Examples

Even or Odd Checker

```
def check(n):
    if n % 2 == 0:
        return "Even"
    else:
        return "Odd"

print(check(10))  # Even
print(check(7))   # Odd
```

Factorial Function

```
def fact(n):
    f = 1
    for i in range(1, n+1):
        f = f * i
    return f

print(fact(5))  # 120
```

10. Things to Remember

- Define a function once, use it many times.
- Use meaningful names for functions and parameters.
- Use return when you need to get a value back from the function.
- Functions can call other functions.
- Keep functions short and focused on one task.

11. Key Takeaways

- A function is defined using the def keyword.
- It can take parameters and return a value.
- You call a function by its name followed by parentheses.
- Functions make code modular, reusable and easy to manage.
- Practice creating and using functions to build better programs!

Good functions make your code DRY (Don't Repeat Yourself) and your programs SMART!

The return Statement and void Function in Python

```python
def add(a, b):
    return a + b

def greet(name):
    print("Hello,", name)
    # No return -> void function
```

Functions can either return a value to the caller using return statement, or perform an action without returning anything (void function).

1. What is return Statement?

- The return statement sends a value from the function back to the caller.
- It can return any data type.
- When return is executed, the function terminates immediately.

2. What is void Function?

- A void function does not return any value to the caller.
- It performs a task (e.g., print, update, calculate) and exits.
- In Python, if a function has no return statement, it returns None by default.

3. The return Statement

Syntax

```python
def function_name(parameters):
    # statements
    return value   # optional
```

Examples

1. Return a value

```python
def square(n):
    return n * n

result = square(5)
print(result)
```

Output: 25

2. Return multiple types

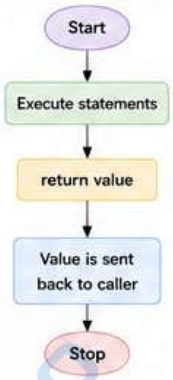
```python
def get_info():
    return "Python", 3.11, "Output:"

lang, ver, status = get_info()
print(lang, ver, status)
```

3. Early return

```python
def check(n):
    if n < 0:
        return "Negative"
    return "Non-negative"

print(check(-10))
print(check(7))
```

Output:
Non-negative

Flow of return



4. void Function (No return Value)

Syntax

```python
def function_name(parameters):
    # statements
    # No return statement
```

Examples

1. Print a message

```python
def greet(name):
    print("Hello,", name)

greet("Alice")
print(greet("Bob"))
```

Output:
Hello, Alice
Hello, Bob
None

2. Perform a task (no return)

```python
def add_and_print(a, b):
    s = a + b
    print("Sum =", s)

add_and_print(4, 6)
```

Output:
Sum = 10

3. Default return value is None

```python
def do_nothing():
    x = 10
    y = 20
    # no return

res = do_nothing()
print(res)
```

Output:
None

Flow of void function

5. When to Use?

Use return when:

- You need to send a result back to the caller.
- You want to use that result for further calculation.
- Example: mathematical functions, getters, validations.

Use void function when:

- You only want to perform an action.
- You don't need to send any value back.
- Example: display messages, logging, updates.

6. return vs void Function

Feature	return Function	void Function
Purpose	Returns a value to the caller	Does not return any value
Syntax	Uses return statement	Has no return statement
Returned Value	Any data type	None (by default)
Use Case	When result is needed	When only an action is needed
Example	`def add(a, b): return a + b`	`def show(): print("Hi")`
Caller	Can store or use the returned value	Cannot store anything useful (it returns None)

7. Common Mistakes

Forgetting return (returns None)

```python
def add(a, b):
    c = a + b   # forgot return

res = add(3, 4)
print(res)   # None
```

Using return outside function

```python
return 10      # SyntaxError
```

Expecting value from void function

```python
def print_msg():
    print("Hi")

x = print_msg()
print(x)     # None
```

Unreachable code after return

```python
def f():
    return 5
    print("This will not run")
```

8. Key Takeaways

- Use return to send a value back from a function.
- If a function has no return, it is a void function.
- Void function returns None by default.
- return stops the function immediately.
- Choose return or void based on the task requirement.
- Good functions are clear, focused and do one thing well.

Pro Tip: In Python, every function returns something.

With return → useful value → Without return → None

"Great programs are built with great functions!"

Scope and Lifetime of Variables in Python

Scope determines where a variable can be accessed in a program.
Lifetime determines how long a variable exists in memory.

```
x = 10          # global scope
def func():
    y = 20      # local to func
    print(x)    # 10  (uses global x)
func()
print(x)        # 10
# print(y)  # Error! y not accessible here
```

1. What is Scope?

- The region of a program where a variable is recognized and can be used.
- Python has four types of scope (**LEGB Rule**).

2. Types of Scope in Python (LEGB Rule)

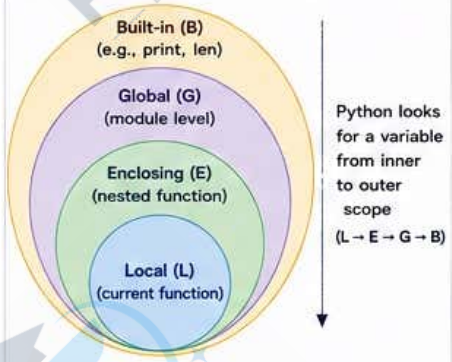
L — Local (function scope)
Inside a function. Accessible only within that function.

E — Enclosing (non-local scope)
Inside nested functions. Refers to the scope of the enclosing function.

G — Global (module scope)
Defined at the top level of a module. Accessible throughout the module.

B — Built-in (Python built-in scope)
Names in the built-in namespace (e.g., len, print, int).

3. Scope Visualization (LEGB)



4. Scope Summary

Scope	Where Declared	Accessible From	Lifetime	Example Variable
Local	Inside a function	Only within that function	Created when the function is called and destroyed when the function ends	`x` inside a function
Enclosing (non-local)	Inside an outer (enclosing) function	Inner nested functions	Exists as long as the enclosing function exists	`y` in `outer()` accessed by `inner()`
Global	At the top level of a module	Throughout the module (after declaration)	From the time the program starts until it ends	`z` at module level
Built-in	Built-in namespace (by Python)	Throughout the program	Exists as long as Python is running	`print`, `len`, `int`

5. Examples

5.1 Local Scope

```
def greet():
    msg = "Hello"
    print(msg)

greet()
# print(msg)    # Error
```

Output:
Hello

Note: msg exists only inside greet().

5.2 Enclosing (Non-local) Scope

```
def outer():
    x = 10      # Enclosing scope
    def inner():
        print(x)  # uses x from outer
    inner()
outer()
```

Output:
10

Note: inner() can access x from enclosing function outer().

5.3 Global Scope

```
x = 5  # Global variable

def show():
    print(x)  # uses global x

show()
print(x)
```

Output:
5
5

Note: x is accessible everywhere in the module.

5.4 Built-in Scope

```
print(len("Python"))  # len is built-in
print(max([3, 7, 1]))  # max is built-in
```

Output:
6
7

Note: Built-in names are always available.

6. Lifetime of Variables

- **Local variables:** Exist only during the execution of the function.
- **Enclosing variables:** Exist as long as the enclosing function is active.
- **Global variables:** Exist from program start until program termination.
- **Built-in variables:** Exist as long as the Python interpreter is running.

Example (Lifetime)

```
def demo():
    a = 100   # exists during function call
    print(a)

demo()
# print(a)  # Error: a no longer exists
```

Output:
100

7. Modifying Variables from Outer Scope

7.1 Using global

```
x = 0

def change():
    global x    # declare x as global
    x = 20      # modify global x

change()
print(x)   # 20
```

Note: global allows modification of global variable inside a function.

7.2 Using nonlocal

```
def outer():
    x = 10

    def inner():
        nonlocal x   # refer to enclosing x
        x = 30

    inner()
    print(x)

outer()   # 30
```

Note: nonlocal allows modification of enclosing variable in nested function.

8. Order of Resolution (LEGB Rule)

Python searches for a variable in this order:

Local (L) → Enclosing (E) → Global (G) → Built-in (B)

The first match is used.

Example:

```
x = "global"

def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print(x)      # local
        print(x)      # enclosing
    inner()
outer()
print(x)              # global
```

Output:
local
enclosing
global

9. Common Mistakes

- Trying to access a local variable outside the function.
- Forgetting global when modifying global variables.
- Forgetting nonlocal when modifying enclosing variables.
- Assuming variables defined in one function are available in another.

10. Key Takeaways

- Scope defines where a variable can be used.
- Lifetime defines how long a variable exists.
- Follow LEGB rule to understand variable resolution.
- Use global or nonlocal to modify outer variables.
- Good understanding prevents bugs and makes code clean and predictable.

Pro Tip: Use the smallest possible scope for your variables. It keeps code clean, safe and easier to understand!

Write better. Think in scopes!

Default Parameters in Python

```
def greet(name, msg="Hello"):
    print(f"{msg}, {name}!")

greet("Alice")              # Hello, Alice!
greet("Bob", "Hi")          # Hi, Bob!
```

Default parameters allow you to define a default value for a parameter in a function.
If no argument is passed for that parameter during the function call, the default value is used.

1 What are Default Parameters?

- Parameters that have default values in the function definition.
- Make functions more flexible and easy to use.
- If an argument is not provided, Python uses the default value.

2 Syntax

```
def function_name(param1, param2=default,
    param3=default, ...):
    # function body
```

3 Rules to Remember

1. Default parameters must be at the end of the parameter list.
2. A parameter with a default value is optional during function call.
3. If you provide a value, the default value is ignored.
4. Default values are evaluated only once, when the function is defined.
5. Use immutable values (int, str, float, tuple, bool) as defaults. Avoid mutable types (list, dict, set) as defaults.

Invalid Example (Rule 1)

Non-default argument follows default argument

```
def wrong(a=10, b):        # X SyntaxError
    pass
# Correct way:
def correct(a, b=10):      #  ✓ Valid
    pass
```

5 Examples

5.1 Basic Default Parameter

```
def greet(name, msg="Hello"):
    print(f"{msg}, {name}!")

greet("Alice")
greet("Bob", "Hi")
```

- msg has default value "Hello".
- First call uses default.
- Second call uses provided value "Hi".

Output:
```
Hello, Alice!
Hi, Bob!
```

5.2 Multiple Default Parameters

```
def add(a, b=0, c=0):
    return a + b + c

print(add(5))
print(add(5, 10))
print(add(5, 10, 15))
```

- b and c have default value 0.
- You can pass 0, 1 or 2 additional arguments.

Output:
```
5
15
30
```

5.3 Default Parameter with Different Data Types

```
def info(name, age=18, city="Unknown"):
    print(f"Name: {name}, Age: {age}, City: {city}")

info("Riya")
info("Rohan", 22)
info("Amit", 25, "Delhi")
```

- age is int, city is str.
- Missing arguments are taken from defaults.

Output:
```
Name: Riya, Age: 18, City: Unknown
Name: Rohan, Age: 22, City: Unknown
Name: Amit, Age: 25, City: Delhi
```

5.4 Keyword Arguments with Defaults

```
def display(a, b=10, c=20):
    print(a, b, c)

display(1)
display(1, c=30)
display(c=40, a=5)
display(a=7, b=8, c=9)
```

- You can override defaults using keyword arguments.
- Order does not matter when using keywords.

Output:
```
1 10 20
1 10 30
5 10 40
7 8 9
```

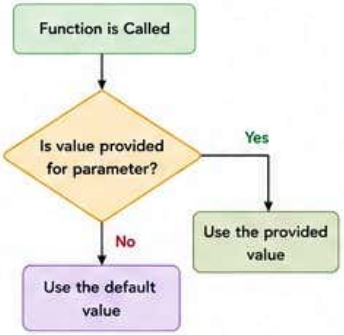
5.5 Default Parameter with Expression

```
import math

def circle_area(r, pi=math.pi):
    return pi * r * r

print(circle_area(2))     # uses default pi
print(circle_area(2, 3.14)) # custom pi
```

- Default value can be an expression.
- You can still override it.

Output:
```
12.566370614359172
12.56
```

6 How Default Parameters Work? (Execution Flow)



Key Points

- Python checks each parameter from left to right.
- If a value is provided, it uses that value.
- If not, it uses the default value defined in the function.
- If no default exists and no argument is provided → Error!

7 Default vs Required Parameters

Feature	Required Parameter	Default Parameter
Definition	No default value	Has a default value
Call Requirement	Must provide value	Optional to provide value
Example	`def f(a):`	`def f(a=10):`
Flexibility	Less flexible	More flexible
Error if missing	Yes	No (default is used)

Common Mistakes

- Putting non-default parameter after default parameter.
- Using mutable default arguments (like `list`, `dict`) → leads to bugs!
- Assuming default value changes during function calls.

Bad Example (Mutable Default)

```
def append_item(item, lst=[]):
    lst.append(item)
    return lst

print(append_item(1))  # [1]
print(append_item(2))  # [1, 2] (why?)
```

9 Key Takeaways

- Default parameters make functions user-friendly.
- They reduce the need for multiple function versions.
- Place default parameters at the end.
- Use immutable values as defaults.
- They improve code readability and flexibility.

Pro Tip Default parameters are evaluated once when the function is defined, not each time the function is called.

Real-life Analogy

Think of default parameters like a restaurant order form.
If you don't choose a spice level, it serves "Medium" by default!

"Good defaults make great functions!"