1. What is Programming?

Definition

Programming is the process of **writing instructions** that a computer follows to perform a task.

Why Programming is Needed

- Computers cannot think on their own
- They only understand instructions
- Programs help automate tasks and solve problems

Real-World Examples

- Calculator → performs calculations
- ATM → processes transactions
- Mobile apps → respond to user actions

Programming Languages

- C, C++, Java, Python, JavaScript
- Python is preferred for beginners because it is **simple and readable**

2. What is Python?

Definition

Python is a **high-level, interpreted, general-purpose programming language**.

Why Python is Beginner-Friendly

- Easy English-like syntax
- No need to write complex code
- Fewer lines compared to other languages

3. Features of Python

- Simple & Easy to Learn
- Interpreted Language
- Platform Independent
- Open Source
- Large Library Support
- Object-Oriented

4. Applications of Python

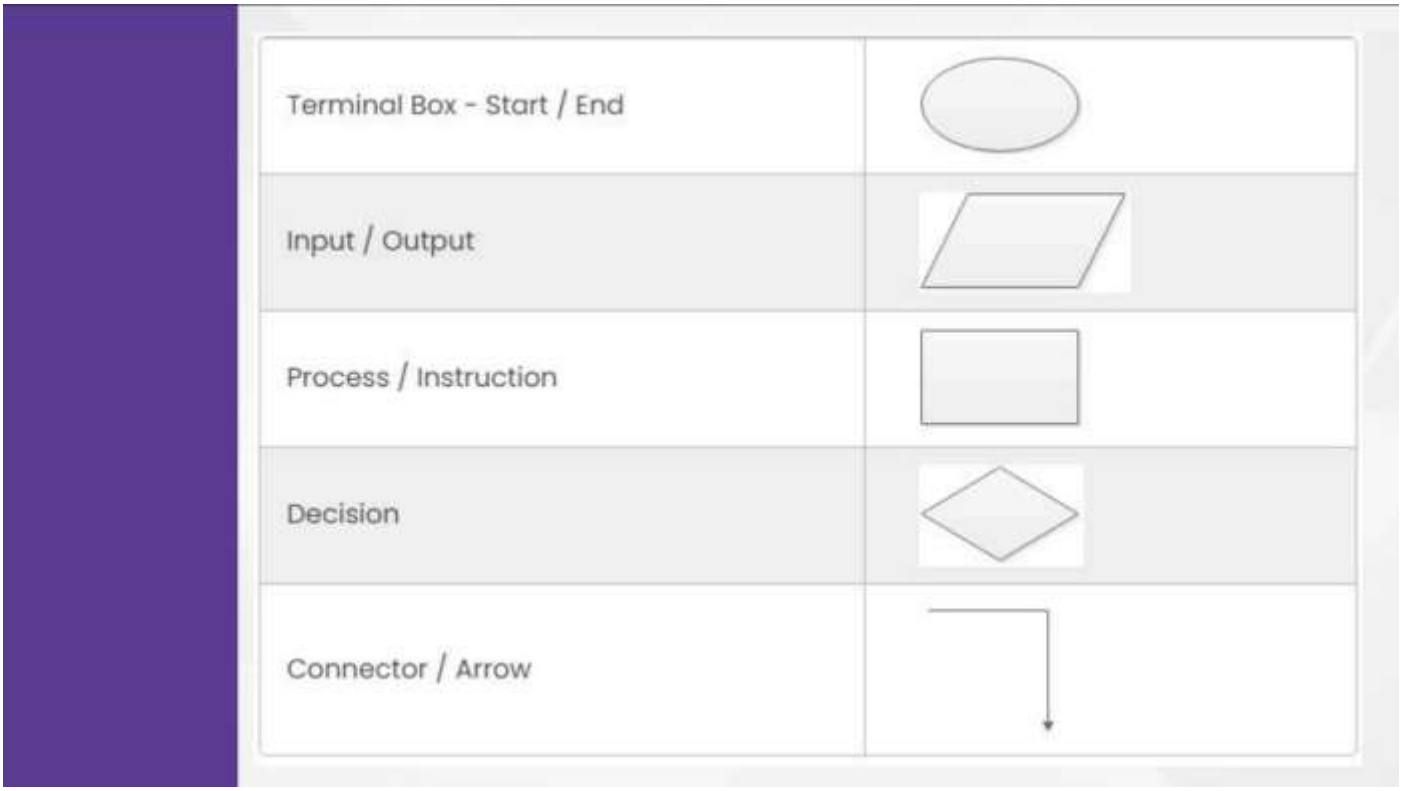
- Data Science & Analytics
- Web Development
- Artificial Intelligence & Machine Learning
- Automation
- Game Development
- Desktop Applications

WHAT IS AN ALGORITHM

- Writing a logical step-by-step method to solve the problem is called the algorithm.
- An algorithm is a procedure for solving problems.
- An algorithm includes calculations, reasoning, and data processing.
- Algorithms can be presented by natural languages, pseudocode, and flowcharts, etc.

WHAT IS A FLOWCHART

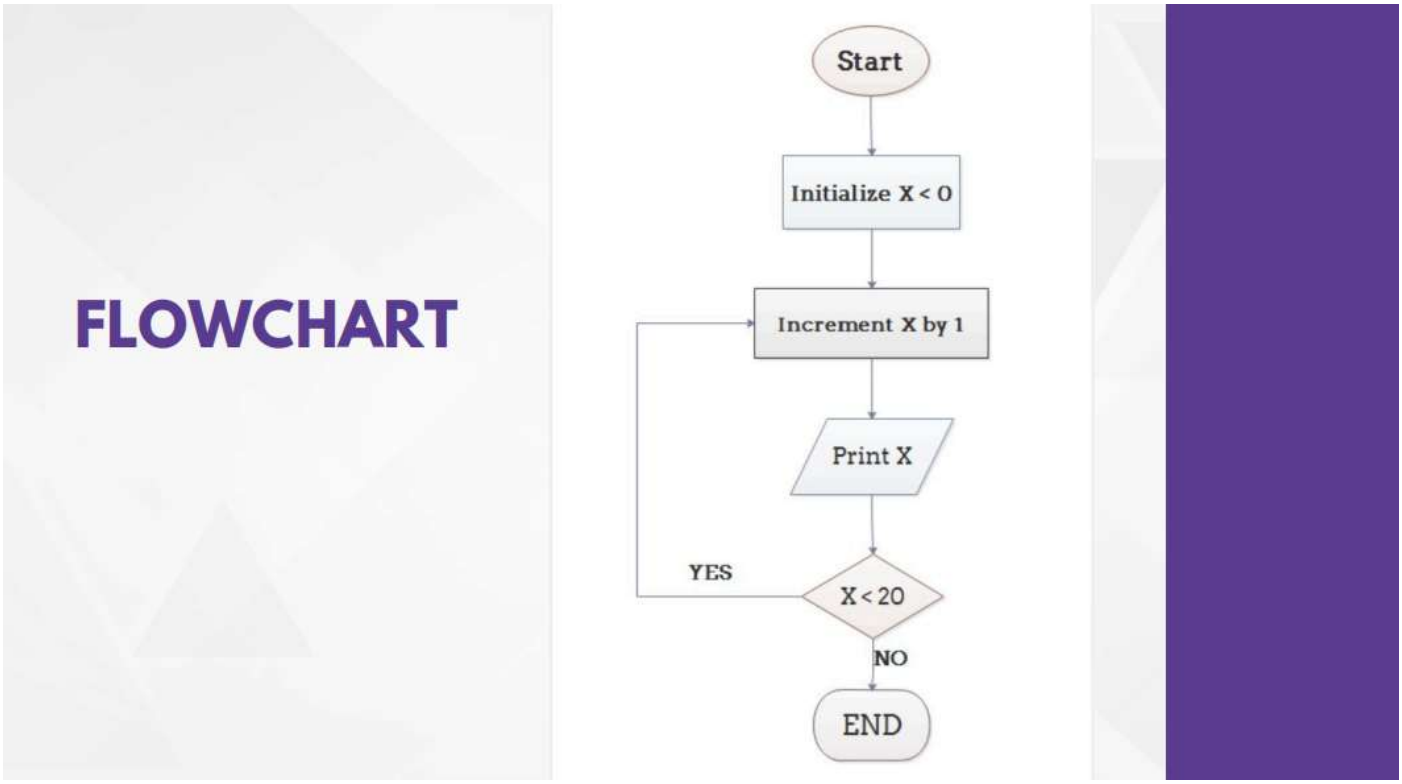
- A flowchart is the graphical or pictorial representation of an algorithm with the help of different symbols, shapes, and arrows to demonstrate a process or a program.
- With algorithms, we can easily understand a program. The main purpose of using a flowchart is to analyze different methods.



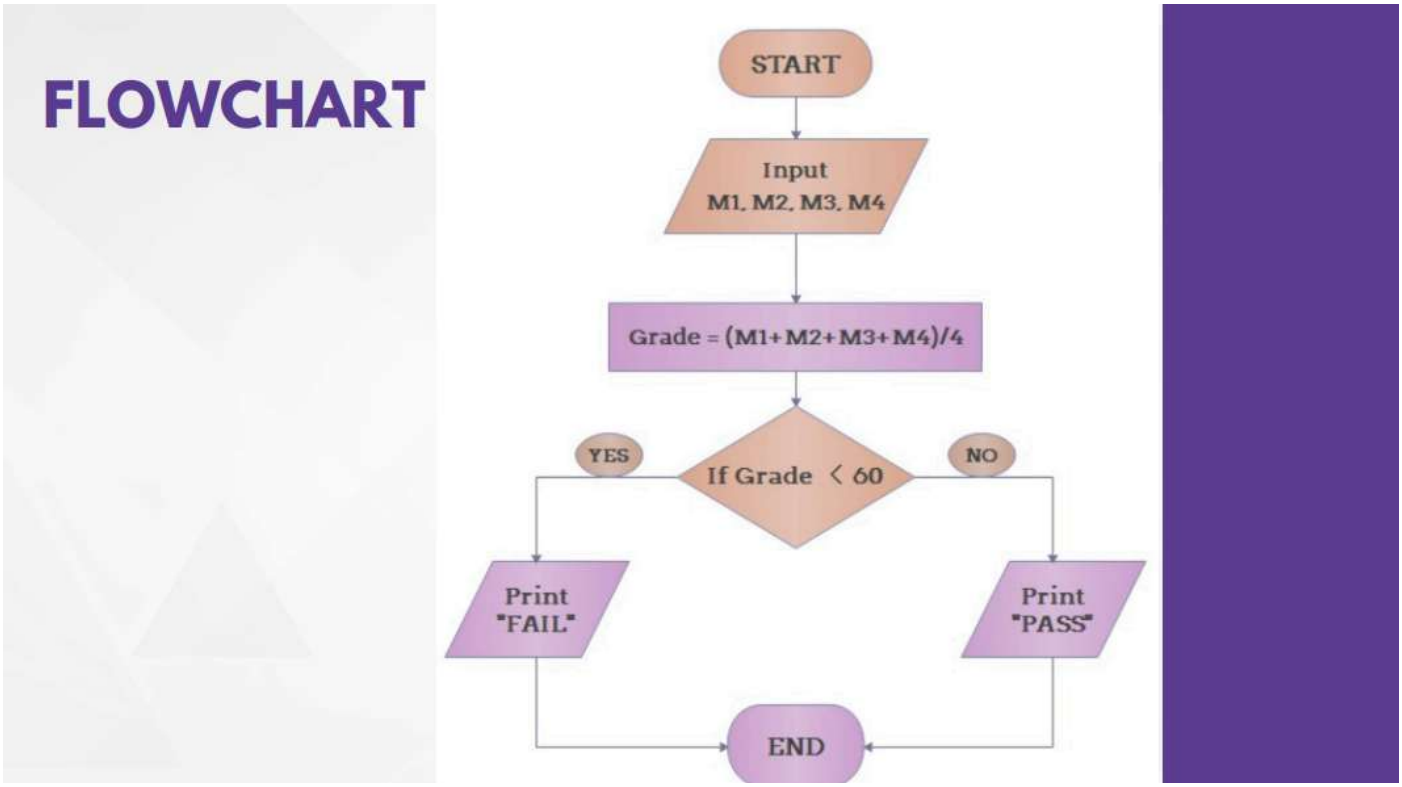
EXAMPLE 1: PRINT 1 TO 20:

ALGORITHM

Step 1: Initialize X as 0,
Step 2: Increment X by 1,
Step 3: Print X,
Step 4: If X is less than 20 then go back to step 2.



EXAMPLE 2:
DETERMINE WHETHER A STUDENT PASSED THE EXAM OR NOT:
ALGORITHM
Step 1: Input grades of 4 courses M1, M2, M3 and M4,
Step 2: Calculate the average grade with formula "Grade=(M1+M2+M3+M4)/4"
Step 3: If the average grade is less than 60, print "FAIL", else print "PASS".



Comments in Python

- Comments are **non-executable lines** used for explanation.
- Comments are **ignored by Python**
- Used to explain code
- Makes program **readable and understandable**

Python supports two types of comments:

- Single Line Comment
- Multi Line Comment

Single Line Comment:
Example:

```
# This is single line comment
```

Multi Line Comment
Example:

```
'''This is
   Multiline
   Comment'''
```

Variables in Python
What is a Variable?

- A variable is a **name given to a value**
- Used to **store data in memory**

- Value of a variable can **change during program execution**

Variable Assignment

- Python uses = for assignment
- No need to declare data type

Example:

x = 10
name = "Python"

Rules for Naming Variables

- Must start with a **letter (a–z, A–Z)** or **underscore (_)**
- Cannot start with a **number**
- Can contain letters, numbers, underscore
- No special characters (@, $, %)
- Cannot use **Python keywords**

Valid:

age = 20
_student = "Python"
marks1 = 85

Invalid:

1name = "A"
total-marks = 90
class = 10

Python is Case-Sensitive

- age and Age are **different variables**

age = 20
Age = 25

Multiple Assignment

Assign same value:

a = b = c = 10

Assign different values:

x, y, z = 1, 2, 3

Dynamic Typing

- Python decides data type **at runtime**
- Variable type can change

x = 10
x = "Python"

Variable Reassignment

- Value of a variable can be updated

count = 5
count = count + 1

Printing Variables

- Variables printed without quotes

name = "Python"
print(name)

Combining Text and Variables

```
age = 21
print("Age is", age)
```

Simple Practice Examples

```
name = "Python"
version = 3.12
print("Language:", name)
print("Version:", version)
```

Data Types in Python

What is a Data Type?

- Data type specifies **what kind of data** a variable holds
- Python determines data type **automatically**
- Checked using type() function

```
x = 10
print(type(x))
```

Classification of Data Types

Python data types are mainly classified into:

1. **Numeric (int, float, complex)**
2. **Text (String)**
3. **Boolean**
4. **Sequence (List, Tuple)**
5. **Set**
6. **Mapping (Dictionary)**

Numeric Data Types

int

- Stores whole numbers
- Positive or negative

```
a = 10
b = -5
```

float

- Stores decimal numbers

```
x = 3.14
y = 10.5
```

complex

- Stores numbers in a + bj format

```
z = 2 + 3j
```

Text Data Type

str

- Stores text or characters
- Written inside quotes

```
name = "Python"
college = 'ABC'
```

Boolean Data Type

bool

- Stores only True or False
- Used in decision making

```
is_valid = True
```

Sequence Data Types

list

- Ordered collection
- Mutable (can change)

```
marks = [80, 85, 90]
```

tuple

- Ordered collection
- Immutable (cannot change)

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

range

- Represents sequence of numbers

```
r = range(1, 5)
```

Set Data Type

set

- Unordered collection
- No duplicate values

```
s = {1, 2, 3}
```

Mapping Data Type

dict

- Stores data in **key–value pairs**

```
student = {"name": "Python", "age": 21}
```

Type Conversion

- Converting one data type to another

```
x = int("10")
y = float(5)
z = str(25)
```

Checking Data Type

- type() function returns data type

```
print(type(10))
print(type("Python"))
```

Simple Practice Examples

```
a = 10
b = 2.5
c = "Hello"
d = True
print(type(a), type(b), type(c), type(d))
```

Programs

1: Check Data Type of Variables
2: Integer Operations
3: Area of a Circle
4: String Data Type
5: String Concatenation
6: Boolean Data Type
7: List Data Type
8: Tuple Data Type
9: Set Data Type
10: Dictionary Data Type

Python Operators:

Python operators are special symbols used to perform specific operations on one or more operands. The variables, values, or expressions can be used as operands.
For example, Python's addition operator (**+**) is used to perform addition operations on two variables, values, or expressions.
The following are some of the terms related to **Python operators**:

- **Unary operators**: Python operators that require one operand to perform a specific operation are known as unary operators.
- **Binary operators**: Python operators that require two operands to perform a specific operation are known as binary operators.
- **Operands**: Variables, values, or expressions that are used with the operator to perform a specific operation.

Types of Python Operators
Python operators are categorized in the following categories −

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators

- Arithmetic operators are used to **perform mathematical calculations**.
- They work with numbers like integers and floats.
- Used for **mathematical calculations**

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)
//	Floor division
**	Exponent (power)

```
a = 10
b = 3
print(a + b)
print(a % b)
```

```
print(a ** b)
```

Comparison Operators

- Comparison operators are used to **compare two values**.
- The result is always a **boolean value (True or False)**.
- Used to **compare two values**
- Result is **True or False**

Operator	Meaning
==	Equal to
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

```
a = 10
b = 20
print(a < b)
```

Logical Operators

- Logical operators are used to **combine multiple conditions**.
- They are commonly used in decision-making statements.
- Used to **combine conditions**

Operator	Meaning
and	True if both are True
or	True if any one is True
not	Reverses the result

```
a = 10
b = 20
print(a < b and b > 15)
```

Assignment Operators

- Assignment operators are used to **assign or update values** in a variable.
- They combine arithmetic operation with assignment.
- Used to **assign values** to variables

Operator	Example
=	x = 5
+=	x += 3
-=	x -= 2
*=	x *= 2
/=	x /= 2

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

Bitwise Operators

- Bitwise operators perform operations at the **binary level**.
- They are mainly used in **low-level programming and optimization**.
- Operate on **binary values**

Operator	Meaning
&	AND
`	`
^	XOR
~	NOT
<<	Left shift
>>	Right shift

```
a = 5
b = 3
print(a & b)
```

Membership Operators

- Membership operators check whether a value **exists in a sequence**.
- Used with lists, strings, tuples, and sets.
- Check whether a value exists in a sequence

Operator	Meaning
in	Present
not in	Not present

```
list1 = [1, 2, 3]
print(2 in list1)
print('a' not in 'python')
```

Identity Operators

- Identity operators check whether two variables **refer to the same memory object**.
- They do not compare values, but **object identity**.
- Check whether two variables refer to **same object**

Operator	Meaning
is	Same object
is not	Different object

```
a = 10
b = 10
print(a is b)
print(a is not b)
```

Python Input and Output (I/O)

What is Input and Output?

- **Input** is the data given to a program by the user.
- **Output** is the result produced by the program.
- Input and output help in making programs **interactive and dynamic**.

Output in Python – print() Function

Purpose of print()

- Used to **display output on the screen**
- Can print text, numbers, variables, and expressions

Basic Syntax

```
print(value)
```

Printing Text
print("Hello Python")

Printing Numbers and Expressions
print(10)
print(5 + 3)

Printing Variables
name = "Python"
print(name)

Printing Multiple Values
- Use commas to separate values

a = 10
b = 20
print("Sum is", a + b)

New Line in Output
- Each print() creates a new line
- \n is used for line break inside a string

print("Hello\nPython")

Custom Separator and End
print("Python", "Java", sep=", ")
print("Welcome", end=" ")
print("User")

Input in Python – input() Function
Purpose of input()
- Used to **take input from the user**
- Input is **always read as a string**

Basic Syntax
variable = input("Message")

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

Type Conversion in Input
Since input is always string, conversion is required.
Integer Input
age = int(input("Enter age: "))
print(age)

Float Input
price = float(input("Enter price: "))
print(price)

Taking Multiple Inputs
Using Separate Statements
a = int(input("Enter a: "))

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

Using split()

```
a, b = map(int, input("Enter two numbers: ").split())
print(a, b)
```

Formatted Output

Using f-strings (Recommended)

```
name = "Python"
age = 21
print(f"Name: {name}, Age: {age}")
```

Input and Output with Expressions

```
a = int(input("Enter number: "))
print("Square:", a * a)
```

Real-World Example: Simple Calculator

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

print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
```

Common Errors in Input & Output

- Forgetting type conversion
- Using quotes incorrectly
- Mixing string and integer without conversion

Error ❌:

```
print("Sum:", "10" + 5)
```

Correct ✔:

```
print("Sum:", int("10") + 5)
```

Python Indexing

What is Indexing?

Indexing in Python is the technique used to **access a single element** from a sequence data type using its **position number (index)**.

Python supports indexing for:

- **Strings**
- **Lists**
- **Tuples**
- **Ranges**

Indexing Syntax

```
sequence[index]
```

- sequence → string / list / tuple
- index → position of the element

Types of Indexing in Python

- Positive Indexing
- Negative Indexing

Positive Indexing

- Starts from **0**
- Moves from **left to right**

Example:

```
text = "PYTHON"
print(text[0])
print(text[3])
```

Output

P
H

Index	**0**	**1**	**2**	**3**	**4**	**5**
Char	P	Y	T	H	O	N

Negative Indexing

- Starts from **-1**
- Moves from **right to left**

Example:

```
print(text[-1])
print(text[-3])
```

Output

N
H

Index	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**
Char	P	Y	T	H	O	N

Indexing in Different Data Types

String Indexing

```
s = "HELLO"
print(s[1])
```

Output

E

Strings are **immutable** → elements cannot be changed.

List Indexing

```
nums = [10, 20, 30, 40]
print(nums[2])
```

Output

30

Lists are **mutable** → values can be modified.

```
nums[1] = 25
print(nums)
```

Output

[10, 25, 30, 40]

Tuple Indexing

```
t = (1, 2, 3, 4)
print(t[-2])
```

Output

3
Tuples are **immutable**, but indexing is allowed.

IndexError
Accessing an invalid index causes an error.
s = "ABC"
print(s[5])
❌ **Error**
IndexError: string index out of range

Slicing in Python

What is Slicing?
Slicing in Python is used to **extract a portion (subsequence)** from sequence data types such as:

- **Strings**
- **Lists**
- **Tuples**
- **Ranges**

It allows accessing multiple elements at once without using loops.

General Syntax of Slicing
sequence[start : stop : step]

Part	Meaning
start	Index to begin slicing (inclusive)
stop	Index to end slicing (exclusive)
step	Interval between elements

👉 All three are **optional**.

Indexing Reminder

- **Positive indexing** starts from 0
- **Negative indexing** starts from -1 (last element)

Example string:
text = "PYTHON"

Index	**0**	**1**	**2**	**3**	**4**	**5**
Char	P	Y	T	H	O	N

Basic Slicing (Start & Stop)
text = "PYTHON"
print(text[1:4])
Output
YTH
Extracts characters from index 1 to 3.

Slicing with Start Only
print(text[2:])
Output
THON
Starts from index 2 till the end.

Slicing with Stop Only

print(text[:4])
Output
PYTH
Starts from beginning till index 3.

Slicing with Step Value
print(text[::2])
Output
PTO
Skips every second character.

Reverse a Sequence Using Slicing
print(text[::-1])
Output
NOHTYP
Negative step reverses the sequence.

Negative Index Slicing
print(text[-5:-2])
Output
YTH
Useful when working from the end of a sequence.

List Slicing Example
nums = [10, 20, 30, 40, 50]
print(nums[1:4])
Output
[20, 30, 40]
Extracts part of a list.

Tuple Slicing Example
t = (1, 2, 3, 4, 5)
print(t[:3])
Output
(1, 2, 3)
Works same as list slicing but tuple is immutable.

Control Statements in Python

What are Control Statements?

- Control statements are used to **control the flow of execution** of a Python program.
- They decide **which statement is executed**, **how many times it is executed**, or **when execution stops**.
- Using control statements, we can write **decision-making, looping, and flow-altering programs**.

Types of Control Statements in Python

Python control statements are broadly classified into **three categories**:

1. **Decision Making Statements**
2. **Looping Statements**
3. **Jump / Control Transfer Statements**

Decision Making Statements

Decision-making statements are used to **execute code based on conditions**.
They depend on **boolean expressions (True / False)**.

1) if Statement

- Executes a block of code **only if the condition is True**.
- If the condition is False, the block is skipped.

Syntax:

```
if condition:
   statement
```

Example:

```
age = 20
if age >= 18:
   print("Eligible to vote")
```

Explanation:

The message is printed only when age >= 18 is True.

2 if–else Statement

- Executes one block if condition is True.
- Executes another block if condition is False.

Syntax:

```
if condition:
   statements
else:
   statements
```

Example:

```
num = 5
if num % 2 == 0:
   print("Even number")
else:
   print("Odd number")
```

3 if–elif–else Statement

- Used to check **multiple conditions**.
- elif means “else if”.

Example:

```
marks = 75
```

```
if marks >= 90:
   print("Grade A")
elif marks >= 60:
   print("Grade B")
else:
   print("Grade C")
```

Use case:
Grading systems, menu-driven programs.

4 Nested if Statement

- An if statement inside another if statement.
- Used for **complex decision-making**.

Example:

```
age = 20
citizen = "India"

if age >= 18:
   if citizen == "India":
      print("Eligible to vote")
```

Looping Statements
Looping statements are used to **execute a block of code repeatedly** as long as a condition is satisfied.

1 for Loop

- Used to iterate over a **sequence** (list, string, range).
- Known number of iterations.

Syntax:

```
for variable in sequence:
   statements
```

Example:

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

Use case:
Tables, list traversal, pattern printing.

2 while Loop

- Executes code **as long as condition is True**.
- Number of iterations is not fixed.

Syntax:

```
while condition:
   statements
```

Example:

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

Use case:
Input validation, menu-based programs.

Jump / Control Transfer Statements
These statements are used to **change the normal flow of loops**.

1 break Statement

- Terminates the loop **immediately**.
- Control moves outside the loop.

Example:

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

Output stops at 4.

2 continue Statement

- Skips the current iteration.
- Continues with the next iteration.

Example:

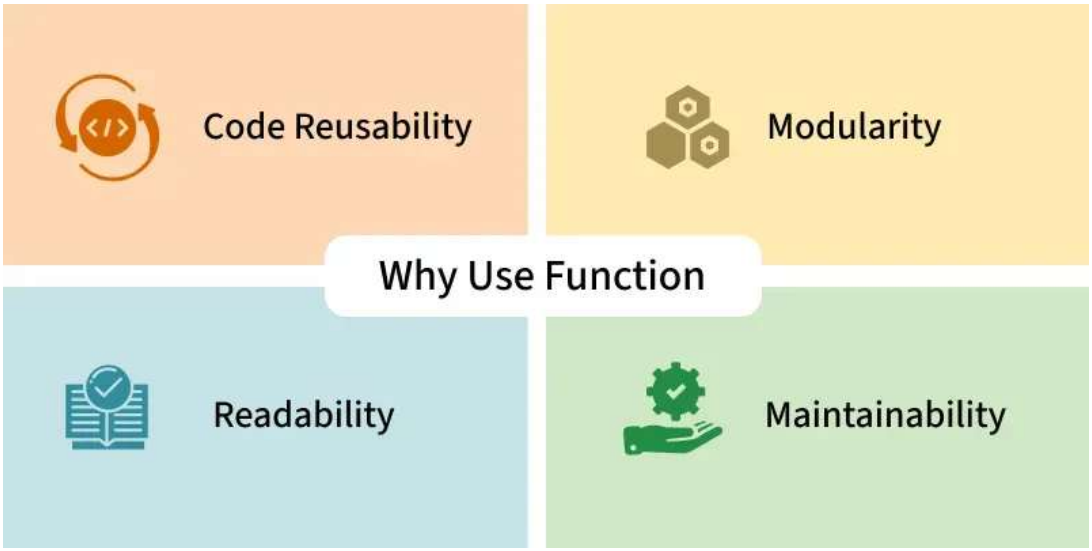
```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Output skips 3.

FUNCTIONS IN PYTHON

What is a Function?

- A function is a **named block of code** that performs a specific task.
- Functions help in **code reusability**, **modularity**, and **easy maintenance**.
- Once defined, a function can be **called multiple times**.



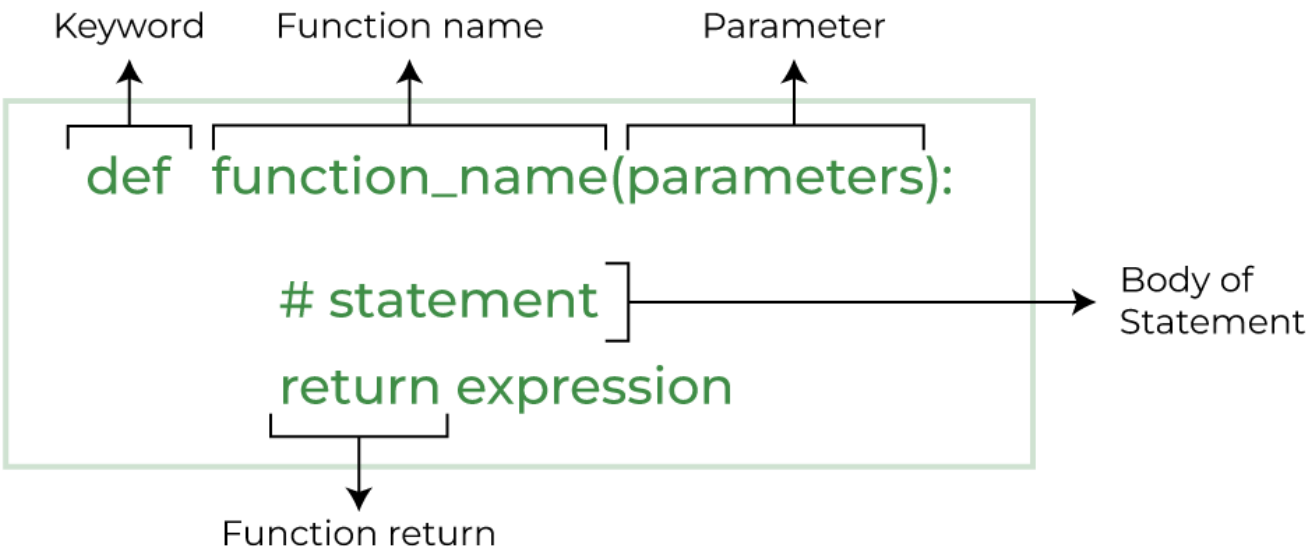
Defining a Function

We can define a function in Python, using the **def** keyword. A function might take input in the form of parameters.

The syntax to declare a function is:

Syntax of a Function

```
def function_name(parameters):
    statements
    return value
```



Calling a Function
After creating a function in Python we can call it by using the name of the functions followed by parenthesis containing parameters of that particular function.

Example of a Simple Function

```
def greet():
   print("Welcome to Python")

greet()
```

Explanation:
The function greet() prints a message when it is called.

Programs:
1. Program to Print a Message "Welcome to Python Programming!"
2. Program to Add Two Numbers
3. Program to Check Even or Odd
4. Program to Print Square of a Number
5. Program to Display Student Details

RETURN VALUES (Fruitful Functions)
What is a Return Value?

- A return value is the **result sent back** by a function using the return keyword.
- Functions that return values are called **fruitful functions**.
- The returned value can be stored in a variable or used in expressions.

Example: Function with Return Value

```
def add(a, b):
   return a + b
print(def(1, 20))
```

Explanation:
The function calculates the sum and returns it to the calling statement.

Returning Multiple Values

```
def calculate(a, b):
   return a + b, a - b
x, y = calculate(10, 5)
```

```
print(x, y)
```

Programs:

1. Program to Add Two Numbers
2. Program to Find Square of a Number
3. Program to Check Even or Odd
4. Program to Find Maximum of Two Numbers
5. Program to Calculate Simple Interest

PARAMETERS AND ARGUMENTS

What are Parameters?

- Parameters are **variables listed in the function definition**.
- They receive values when the function is called.

What are Arguments?

- Arguments are the **actual values passed** to a function during a call.

Example

```
def square(n):   # n is parameter
   return n * n

print(square(5))  # 5 is argument
```

Types of Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following function argument types in Python, Let's explore them one by one.

1. Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

```
def myFun(x, y=50):
   print("x: ", x)
   print("y: ", y)
myFun(10)
```

Output

```
x:  10
y:  50
```

2. Keyword Arguments

In keyword arguments, values are passed by explicitly specifying the parameter names, so the order doesn’t matter.

```
def student(fname, lname):
   print(fname, lname)
student(fname='Python', lname='Practice')
student(lname='Practice', fname='Python')
```

Output

```
Python Practice
Python Practice
```

3. Positional Arguments
In positional arguments, values are assigned to parameters based on their order in the function call.

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)

print("Case-1:")
nameAge("Suraj", 27)

print("\nCase-2:")
nameAge(27, "Suraj")
```

Output
Case-1:
Hi, I am Suraj
My age is 27

Case-2:
Hi, I am 27
My age is Suraj

Recursive Functions
A recursive function is a function that calls itself to solve a problem. It is commonly used in mathematical and divide-and-conquer problems. Always include a base case to avoid infinite recursion.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(4))
```

LOCAL AND GLOBAL SCOPE
What is Scope?

- Scope defines **where a variable can be accessed** in a program.
- Python mainly supports **local** and **global** scope.

Local Scope

- A variable declared **inside a function** is called a local variable.
- It is accessible **only within that function**.

Example:

```
def show():
    x = 10   # local variable
    print(x)
show()
```

x cannot be used outside the function.

Global Scope

- A variable declared **outside all functions** is called a global variable.
- It can be accessed anywhere in the program.

Example:

```
x = 20
```

```
def display():
   print(x)
display()
```

Using Global Variable Inside a Function

```
x = 5
def update():
   global x
   x = 10
update()
print(x)
```

Explanation:

The global keyword allows modification of a global variable inside a function.

Difference Between Local and Global Variables

Feature	Local Variable	Global Variable
Scope	Inside function only	Entire program
Lifetime	Function execution	Program execution
Access	Limited	Wide

STRINGS IN PYTHON

What is a String?

In Python, a string is a sequence of characters written inside quotes. It can include letters, numbers, symbols, and spaces.

- Python does not have a separate character type.
- A single character is treated as a string of length one.
- Strings are commonly used for text handling and manipulation.

Creating a String

Strings can be created using either **single ('...')** or **double ("...")** quotes. Both behave the same.

Example: Creating two equivalent strings one with single and other with double quotes.

```
s1 = 'Python'   # single quote
s2 = "Python"   # double quote
print(s1)
print(s2)
```

Multi-line Strings

Use triple quotes **('''...''')** or **("""...""")** for strings that span multiple lines. Newlines are preserved.

Example: Define and print multi-line strings using both styles.

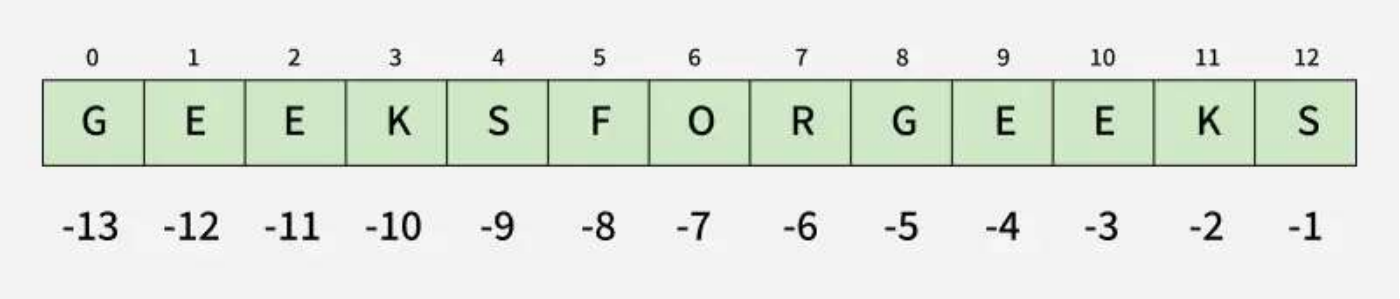
```
s = """I am Learning
Python String"""
print(s)
```

String Indexing

- Index starts from **0** (left to right).
- Negative indexing starts from **–1** (right to left).

Accessing characters in String

Strings are indexed sequences. Positive indices start at **0** from the **left**; negative indices start at **-1** from the **right** as represented in below image:




```
s = "Python Programming"
print(s[0])   # first character
print(s[4])   # 5th character

s = " Python Programming "
print(s[-10])   # 3rd character
print(s[-5])    # 5th character from end
```

String Slicing

Slicing is a way to extract a portion of a string by specifying the **start** and **end** indexes. The syntax for slicing is **string[start:end:]**, where **start** starting index and **end** is stopping index (excluded).

Example: In this example we are slicing through range and reversing a string.

```
s = "Python Programming"
print(s[1:4])    # characters from index 1 to 3
print(s[:3])     # from start to index 2
print(s[3:])     # from index 3 to end
print(s[::-1])   # reverse string
```

String Iteration

Strings are iterable; you can loop through characters one by one.

Example: Here, it print each character on its own line.

```
s = "Python"
for char in s:
    print(char)
```

STRING SLICING

What is String Slicing?

- String slicing is used to **extract a portion of a string**.
- It does not modify the original string.

STRING FUNCTIONS AND METHODS

Built-in String Functions

These functions work on strings.

```
s = "Python"
print(len(s))     # Length of string
print(max(s))     # Maximum character
print(min(s))     # Minimum character
```

Common String Methods

String methods are **called using dot (.) operator**.

Case Conversion Methods

```
text = "python programming"
print(text.upper())
print(text.lower())
print(text.title())
print(text.capitalize())
```

Searching and Checking Methods

```
s = "Python123"

print(s.isalpha())
print(s.isdigit())
print(s.isalnum())
print(s.startswith("Py"))
print(s.endswith("on"))
```

String Immutability Example

```
s = "Python"
# s[0] = "J"  ❌ Error
```

✔ Strings cannot be modified directly.

Real-World Example: Palindrome Check

```
word = input("Enter word: ")

if word == word[::-1]:
   print("Palindrome")
else:
   print("Not Palindrome")
```

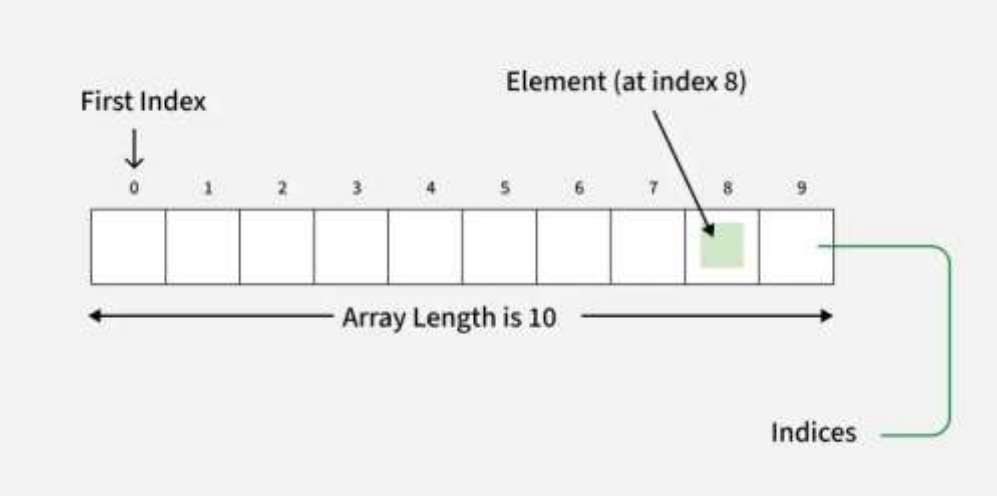
Difference: String Functions vs Methods

Feature	Functions	Methods
Usage	Function(string)	string.method()
Example	len(s)	s.upper()

PYTHON ARRAYS

What is an Array?

- An array is a data structure used to **store multiple values of the same data type**.
- In Python, arrays are provided using the **array module**.
- Arrays are more **memory efficient** than lists when storing same-type elements.



Importing Array Module
Before creating an array, we must import the array module.
from array import array

Creating an Array in Python
Syntax
array(typecode, elements)

- **typecode** specifies the data type
- **elements** are values stored in the array

Common Type Codes

- 'I' → integer
- 'f' → float
- 'u' → string

Example: Integer Array

```
from array import array
arr = array('i', [10, 20, 30, 40])
print(arr)
```

Accessing the Elements of an Array

- Array elements are accessed using **index numbers**.
- Indexing starts from **0**.
- Negative indexing is also supported.

Example: Accessing Elements

```
print(arr[0])   # First element
print(arr[2])   # Third element
print(arr[-1])  # Last element
```

Traversing an Array
Elements can be accessed using loops.
Using for Loop

```
for x in arr:
   print(x)
```

Modifying Array Elements

- Array values can be updated using index.

```
arr[1] = 25
print(arr)
```

ARRAY METHODS
Python arrays support several built-in methods to manipulate elements.

append() – Add element at the end

```
arr.append(50)
print(arr)
```

insert() – Insert element at specific position

```
arr.insert(1, 15)
print(arr)
```

remove() – Remove first occurrence of an element

arr.remove(30)
print(arr)

pop() – Remove last element
arr.pop()
print(arr)

index() – Find index of an element
print(arr.index(20))

count() – Count occurrences of an element
arr.append(20)
print(arr.count(20))
reverse() – Reverse array elements
arr.reverse()
print(arr)

Length of an Array

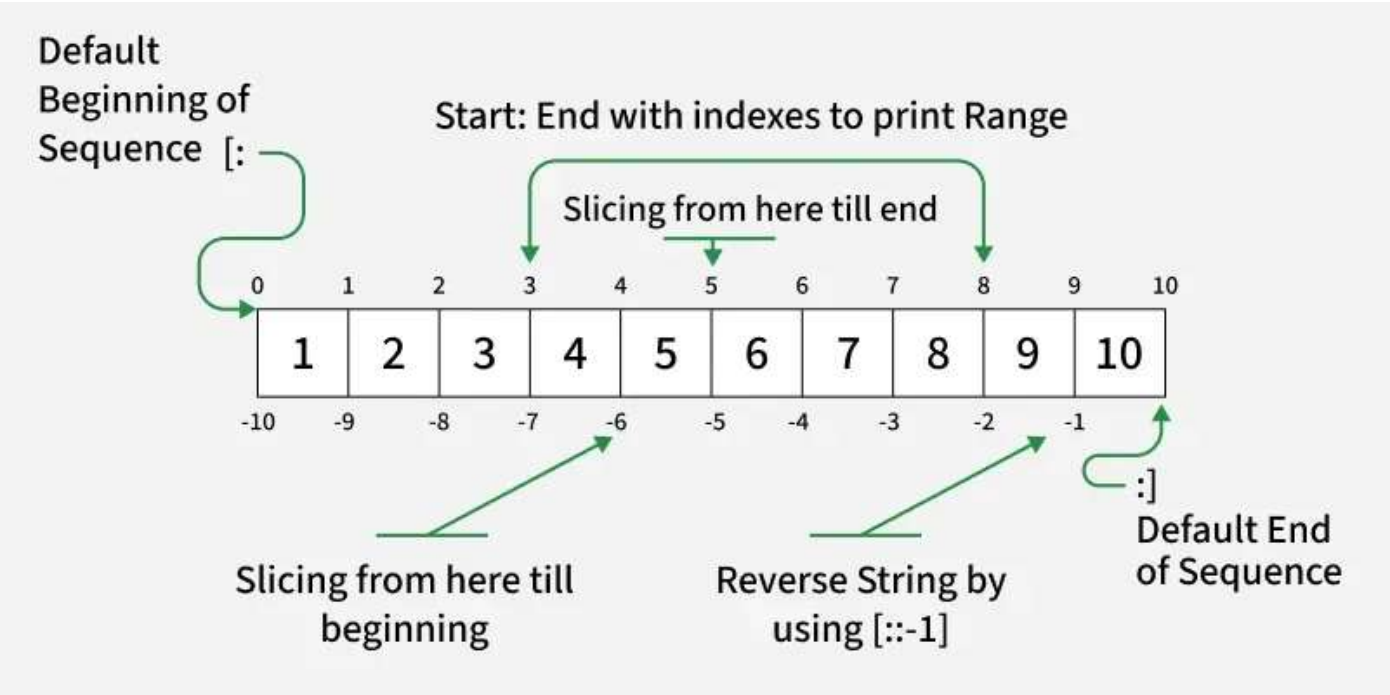
- Use len() function to find number of elements.

print(len(arr))

Difference Between Array and List

Feature	Array	List
Data Type	Same type	Mixed types
Module Required	Yes	No
Memory Efficient	Yes	Less
Speed	Faster	Slower

Slicing of an Array
In Python array, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use Slice operation.



- Elements from beginning to a range use [:Index]
- Elements from end use [:-Index]

- Elements from specific Index till the end use [Index:]
- Elements within a range, use [Start Index:End Index]
- Print complete List, use [:].
- For Reverse list, use [::-1].

```
import array as arr
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = arr.array('i', a)
res = a[3:8]
print(res)
res = a[5:]
print(res)
res = a[:]
print(res)
```

Searching Element in an Array

In order to search an element in the array we use a python in-built index() method. This function returns the index of the first occurrence of value mentioned in arguments.

```
import array
a = array.array('i', [1, 2, 3, 1, 2, 5])
# index of 1st occurrence of 2
print(a.index(2))
# index of 1st occurrence of 1
print(a.index(1))
```

Real-World Example: Sum of Array Elements

```
from array import array
arr = array('i', [10, 20, 30, 40])
total = 0
for i in arr:
    total += i
print("Sum:", total)
```

FILE HANDLING IN PYTHON

What is File Handling?

- File handling allows a program to **store data permanently** on secondary storage.
- Python provides built-in functions to **create, read, write, and modify files**.
- Files help in **data persistence**, unlike variables which store data temporarily.

Types of Files in Python

1. **Text files** – .txt, .csv
2. **Binary files** – .bin, .dat

Opening a File in Python

- Files are opened using the **open()** function.

Syntax:

```
file = open("filename", "mode")
```

Closing a File

- Files should be closed after use to **free system resources**.

file.close()

WORKING WITH CSV FILES (.csv)

What is a CSV File?

- CSV stands for **Comma Separated Values**.
- Used to store **tabular data**.
- Each row is a record, each column is a field.

CSV Module

Python provides the **csv module** to work with CSV files.

import csv

Reading from a CSV File

```
import csv
with open("students.csv", "r") as file:
   reader = csv.reader(file)
   for row in reader:
      print(row)
```

EXCEPTION HANDLING IN PYTHON

What is an Exception?

- An exception is a **runtime error** that occurs during program execution.
- When an exception occurs, the **normal flow of the program is interrupted**.
- If not handled, the program **terminates abruptly**.

What is Exception Handling?

- Exception handling is the process of **detecting and managing runtime errors**.
- Python uses **try, except, else, and finally** blocks to handle exceptions.
- It helps in **preventing program crashes** and improving reliability.

Why Exception Handling is Required?

- To avoid sudden termination of programs
- To display user-friendly error messages
- To handle unexpected input or system errors
- To maintain smooth program execution

Structure of Exception Handling

```
try:
   # code that may cause an exception
except ExceptionType:
   # code to handle exception
else:
   # code executed if no exception occurs
finally:
   # code executed in all cases
```

try Block

- The try block contains **code that may produce an error**.
- Python checks this block for exceptions.

```
try:
   x = int(input("Enter number: "))
   print(x)
```

except Block

- The except block handles the **exception raised in try block**.
- Prevents abnormal termination.

```
except ValueError:
   print("Invalid input! Enter a number.")
```

else Block

- The else block executes **only when no exception occurs**.
- Used for code that should run **after successful execution**.

```
else:
   print("Input accepted successfully")
```

finally Block

- The finally block executes **always**, whether exception occurs or not.
- Used for **cleanup operations** like closing files.

```
finally:
   print("Execution completed")
```

Example: Complete Exception Handling

```
try:
   a = int(input("Enter a number: "))
   b = int(input("Enter another number: "))
   print("Result:", a / b)
except ValueError:
   print("Please enter valid integers")
except ZeroDivisionError:
   print("Division by zero is not allowed")
else:
   print("Calculation successful")
finally:
   print("Program ended")
```

COMMON EXCEPTIONS IN PYTHON

1. ValueError

- Occurs when a function receives **correct type but invalid value**.
- Common in input type conversion.

Example:

```
try:
   x = int("abc")
except ValueError:
   print("ValueError occurred")
```

Reason: Cannot convert "abc" to integer.

2. TypeError

- Occurs when an operation is applied to **incompatible data types**.

Example:

```
try:
   a = 10 + "5"
except TypeError:
   print("TypeError occurred")
```

Reason: Cannot add integer and string.

3. FileNotFoundError

- Occurs when attempting to **open a file that does not exist**.

Example:

```
try:
   file = open("data.txt", "r")
except FileNotFoundError:
   print("File not found")
```

Multiple Except Blocks

- Used to handle different exceptions separately.

```
try:
   x = int(input("Enter number: "))
   print(10 / x)
except ValueError:
   print("Invalid input")
except ZeroDivisionError:
   print("Cannot divide by zero")
```

Difference Between Error and Exception

Error	Exception
Compile-time or logical	Runtime
Cannot be handled	Can be handled
Stops execution	Can continue

OOPS IN PYTHON

What is OOPS?

- Object-Oriented Programming System (OOPS) is a programming approach based on **objects and classes**.
- It helps to organize programs by **modeling real-world entities**.
- Python fully supports object-oriented programming.

Advantages of OOPS

- Code reusability
- Better data security
- Easy maintenance
- Modularity and scalability

Main OOPS Concepts in Python

1. Class
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction

Class

What is a Class?

- A class is a **blueprint or template** for creating objects.
- It defines **data (variables)** and **functions (methods)**.

Syntax

```
class Student:
   name = "Python"
```

Object

What is an Object?

- An object is an **instance of a class**.
- It represents a **real-world entity**.

Example

```
class Student:
   name = "Python"

s1 = Student()
print(s1.name)
```

Encapsulation

What is Encapsulation?

- Encapsulation is the process of **binding data and methods together**.
- It helps in **data hiding and security**.

Example

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

   def show_balance(self):
      print("Balance:", self.balance)

b = Bank(5000)
b.show_balance()
```

Note:

Variables prefixed with _ or __ are treated as protected/private by convention.

Inheritance

What is Inheritance?

- Inheritance allows one class to **inherit properties and methods** of another class.
- It promotes **code reuse**.

Types of Inheritance

- Single
- Multiple
- Multilevel
- Hierarchical

Example: Single Inheritance

```
class Parent:
   def show(self):
      print("Parent class")
```

```
class Child(Parent):
    def display(self):
        print("Child class")

c = Child()
c.show()
c.display()
```

Polymorphism

What is Polymorphism?

- Polymorphism means **one name, many forms**.
- Same method name behaves **differently** in different contexts.

Example: Method Overriding

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

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

d = Dog()
d.sound()
```

Abstraction

What is Abstraction?

- Abstraction hides **implementation details** and shows **essential features**.
- Achieved using **abstract classes**.

Example Using abc Module

```
from abc import ABC, abstractmethod

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

class Square(Shape):
    def area(self):
        print("Area of square")

s = Square()
s.area()
```

Constructor (__init__ Method)

What is a Constructor?

- A constructor is a special method that is **automatically called** when an object is created.
- Used to **initialize object data**.

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

```
s = Student("Python")
print(s.name)
```

self Keyword

- self represents the **current object**
- Used to access instance variables and methods.

Difference Between Class and Object

Class	**Object**
Blueprint	Instance
Logical entity	Physical entity
No memory allocation	Memory allocated

Difference Between OOPS and Procedure-Oriented Programming

OOPS	**POP**
Based on objects	Based on functions
Data security	Less security
Reusability	Less reuse