
1010101010
0101010101
1010101010
0101010101
Notes
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
Good Notes
Better Understanding
Better Results


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;
}

Introduction:

History of Python Programming Language

- Python programming language was created by Guido van Rossum. He started working on Python in the late 1980s.
- Python was first released in February 1991. It was designed to be easy to read, easy to use, and powerful.
- The name "Python" was inspired by the British comedy group "Monty Python's Flying Circus", which Guido van Rossum was a big fan of.
- Python 1.0 was released in January 1994, with many new features and improvements.
- Over the years, Python has evolved through many versions. Some major versions are Python 2.0 (2000), Python 3.0 (2008).
- Today, Python is one of the most popular programming languages in the world. It is widely used in web development, data science, artificial intelligence, automation, and many other fields.

Thrust area of Python

Python is a versatile programming Language and it has a strong presence in many thrust areas.

1. Web Development : Python is widely used for web development using frameworks like Django, Flask, Pyramid, etc. It helps in building dynamic and powerful websites quickly.
2. Data Science and Analytics : Python is a top choice for data analysis, data visualization and machine learning. Libraries like NumPy, Pandas, Matplotlib, Scikit-learn are widely used in this area.
3. Artificial Intelligence and Machine Learning : Python supports AI and ML with libraries such as TensorFlow, Keras, PyTorch. It is used to build smart systems that can learn and make decisions.
4. Automation and Scripting : Python is used for automating repetitive tasks like file handling, web scraping, testing, and system administration.
5. Game Development : Python is used in game development using libraries like Pygame. It allows easy creation of 2D games and simple interactive applications.
6. Scientific and Numeric Computing : Python is used in scientific research, simulations and engineering applications. Libraries like SciPy and SymPy are commonly used in this field.

Installing Anaconda Python Distribution

Anaconda is a popular Python distribution that comes with Python interpreter, Jupyter Notebook and many useful libraries for data science, machine learning and scientific computing.

Steps to Install Anaconda :

1. **Visit the Official Website :**
   Go to the Anaconda official website : https://www.anaconda.com and click on the "Download" button.
2. **Choose the Installer :**
   Select the appropriate installer for your operating system (Windows, macOS or Linux).
3. **Download the Installer :**
   Click on the "Download" button for Anaconda Individual Edition. The installer file will be downloaded.
4. **Run the Installer :**
   Locate the downloaded installer file and double-click on it to start the installation process.
5. **Follow the Installation Wizard :**
   Follow the on-screen instructions. You can keep the default settings and click "Next".
6. **Complete the Installation :**
   Click "Install" and wait for the process to complete. Once installed, click "Finish".
7. **Verify the Installation :**
   Open Anaconda Navigator from the Start Menu (Windows) or Applications folder (macOS). If it opens successfully, Anaconda has been installed.

> Anaconda Navigator provides an easy interface to manage environments, packages and launch tools like Jupyter Notebook, Spyder, etc.

Installing PyCharm IDE to setup a Python Development Environment

PyCharm is a popular Integrated Development Environment (IDE) used for Python development. It provides smart coding assistance, debugging, project management and many other useful features. Follow the steps below to install PyCharm and set up your Python development environment.

Steps to Install PyCharm IDE :

1. **Visit the Official Website :**
   - Go to the official JetBrains PyCharm website : https://www.jetbrains.com/pycharm/
   - Click on the "Download" button.
2. **Choose the Edition :**
   - You can choose between PyCharm Community Edition (Free) and PyCharm Professional Edition (Paid). For beginners, the Community Edition is recommended.
3. **Download the Installer :**
   - Click on the Download button for your operating system (Windows, macOS or Linux). The installer file will be downloaded.
4. **Run the Installer :**
   - Locate the downloaded installer file and double-click on it to start the installation.
5. **Follow the Setup Wizard :**
   - Follow the on-screen instructions. Accept the license agreement, choose the installation location and click "Next".
6. **Configure Installation Options :**
   - Choose options such as creating desktop shortcuts and adding PyCharm to the system PATH (recommended). Then click "Install".
7. **Complete the Installation :**
   - Once the installation is complete, click "Finish".
8. **Launch PyCharm :**
   - Open PyCharm from the Start Menu (Windows) or Applications folder (macOS).
   - On the first launch, you can choose to import settings or start with default settings.
9. **Set Up Python Interpreter :**
   - Go to File → Settings (or PyCharm → Preferences on macOS) → Project: <your project> → Python Interpreter.
   - Click the gear icon → Add Interpreter → select "Existing environment" or "New environment" → choose your Python interpreter → OK.
   - Now your Python development environment is ready to use !

PyCharm provides powerful tools for coding, debugging, testing and project management, making Python development faster and more efficient.

Creating and running your first Python Project

Let's create a simple "Hello, World!" program in PyCharm and run it. This will be your first Python project.

Steps :

1. **Create a New Project :**
   - Open PyCharm IDE.
   - Click on "New Project".
   - Give a project name, for example: MyFirstProject
   - Choose a location to save the project.
   - Select the Python interpreter (default is fine) and click "Create".
2. **Create a Python File :**
   - In the project window, right-click on your project name.
   - Go to New → Python File.
   - Name the file: main.py and click OK.
3. **Write Your First Python Program :**
   - Open main.py and type the following code :

```
# This is my first Python program
print("Hello, World!")
```

← This program prints Hello, World! on the screen.

4. **Run the Program :**
   - Click on the green ▶ Run button on the top right corner.
   - Or right-click on the code editor → Run 'main'.
   - The output will appear in the Run tool window.
5. **Output :**

```
Run:  main  ×
C:\Users\user\PycharmProjects\MyFirstProject\venv\Scripts\python.exe
C:\Users\user\PycharmProjects\MyFirstProject\main.py
Hello, World!

Process finished with exit code 0
```

Congratulations! You have created and run your first Python project successfully. Keep exploring and happy coding! ☺

Parts of Python Programming Language:

1. Identifiers

- Identifiers are the names given to variables, functions, classes, modules or any other user-defined objects in a Python program.
- They help to identify different elements in the program.

Rules for Identifiers :

1. An identifier can contain letters (a-z, A-Z), digits (0-9) and underscore (_).
2. An identifier must start with a letter (a-z, A-Z) or underscore (_).
3. An identifier cannot start with a digit.
4. Identifiers are case-sensitive. (myVar and myvar are different)
5. No special symbols are allowed except underscore (_).
6. Python keywords cannot be used as identifiers.

Examples :

Valid Identifiers			Invalid Identifiers		
name	_name	name1	1name	name!	my var
student_name		sum1	for	while	2value
age_2024		_value	class	@data	sum-1
myVariable		TOTAL			

2. Keywords

- Keywords are reserved words in Python that have a special meaning.
- They have predefined purposes and cannot be used as identifiers (variable names, function names, etc.).
- All keywords in Python are written in lowercase.

List of Python Keywords :

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

Statements

In Python, a statement is an instruction that the Python interpreter can execute. Each statement performs some action. Python statements are written on a single line or can span multiple lines using line continuation.

1. Simple Statements :

These are the most basic statements and perform a simple action.

Examples :

- Assignment statement → `x = 10` # Assignment
- Expression statement → `x + y` # Expression
- Print statement → `print("Hello, Python!")` # Print
- Delete statement → `del x` # Delete
- Pass statement → `pass` # Pass (does nothing)

2. Compound (Structured) Statements :

These statements contain one or more blocks of statements.

Examples :

- If statement
- If...else statement
- For loop
- While loop
- Function definition
- Class definition
- Try...except statement

```
if x > 0:
    print("Positive")                    # If statement
if x > 0:
    print("Positive")
else:
    print("Non-positive")                # If...else statement
for i in range(5):
    print(i)                             # For loop
while x > 0:
    x = x - 1                            # While loop
def add(a, b):
    return a + b                         # Function definition
class Student:
    pass                                 # Class definition
try:
    x = 10 / y
except ZeroDivisionError:
    print("Cannot divide by zero")       # Try...except statement
```

3. Multi-line Statements :

A statement can extend over multiple lines using line continuation ( ).

Example :

```
total = 1 + 2 + 3 + \
        4 + 5 + 6                        # Line continuation using backslash
print(total)                             # Output: 21
```

Expressions

An expression in Python is a combination of values, variables, operators and function calls that is evaluated to produce a single result (a value). Expressions are used in computations, assignments, conditions, function calls, etc.

Components of an Expression :

- Operands → Values or variables on which operations are performed.
- Operators → Symbols that perform operations on operands.
- Parentheses → Used to group sub-expressions and control precedence.

Examples of Expressions :

```
①  5 + 3                  →  8                          # Arithmetic expression
②  x * y                  →  product of x and y         # Using variables
③  (a + b) / 2            →  average of a and b         # Using parentheses
④  x**2 + 3*x + 2         →  polynomial expression      # Using exponent and operators
⑤  len(name)              →  length of string "name"    # Function call expression
⑥  a > b                  →  True or False              # Relational expression
⑦  (x > 0) and (y < 5)    →  True or False              # Logical expression
```

Types of Expressions :

Type	Description	Example	Evaluates to
Arithmetic Expression	Uses arithmetic operators (+, -, *, /, %, //, **)	10 + 4 * 2	18
Relational Expression	Uses relational operators (==, !=, <, >, <=, >=)	x >= 10	True / False
Logical Expression	Uses logical operators (and, or, not)	(x > 0) and (y < 5)	True / False
Assignment Expression	Assigns a value to a variable	x = 5 + 3	8 (assigned to x)
Function Call Expression	Calls a function and returns a value	max(a, b)	Maximum of a and b

Rules :

- An expression must have at least one operand.
- The order of evaluation is determined by operator precedence.
- Parentheses can be used to change the default order.

Example in Python Shell :

```
>>> x = 10              # Statement (assignment)
>>> y = 3
>>> x + y               # Expression
    13
>>> (x * y) + 2         # Expression
    32
>>> x > y               # Expression
    True
```

More Examples :

```
>>> 2 ** 3                  # Exponent expression
    8
>>> (5 + 2) * (8 - 3)       # Arithmetic expression
    35
>>> not (x == y)            # Logical expression
    True
```

Variables

A variable is a name given to a memory location in a program. It is used to store data (values) that can be changed during the execution of a program.

Key Points :

- Variables act as containers for storing data values.
- The value stored in a variable can change.
- Python is a dynamically typed language, so you do not need to declare the type of variable.
- A variable is created automatically when you assign a value to it.

Syntax :

```
variable_name = value
```

variable_name → name of the variable
value → data stored in the variable

Example :

```
x = 10              # integer variable
name = "Python"     # string variable
price = 25.5        # float variable
is_valid = True     # boolean variable
```

Rules for Naming Variables :

1. Must start with a letter (a-z, A-Z) or underscore (_).
2. Can contain letters, digits (0-9) and underscore (_).
3. Cannot start with a digit.
4. No spaces or special characters allowed other than underscore (_).
5. Variable names are case-sensitive (age, Age and AGE are different).

Examples of Valid and Invalid Variables :

Valid Variables	Invalid Variables	
age	1age	(starts with digit)
_name	total marks	(contains space)
total_marks	@name	(contains special character)
student1	for	(keyword cannot be used)
Price2	stu$dent	(contains special character)

Changing the Value of a Variable :

```
x = 5               # x stores 5
x = x + 2           # x now stores 7
x = 20              # x now stores 20
```

Multiple Variables in One Line :

```
a, b, c = 1, 2, 3       # a = 1, b = 2, c = 3
x = y = z = 100         # x = 100, y = 100, z = 100
```

Operators

Operators are special symbols that perform operations on one or more operands (values or variables). Python supports a wide range of operators.

1. Types of Operators in Python :

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

2. Arithmetic Operators : Used to perform mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 3	1.666...
//	Floor Division (Quotient)	5 // 3	1
%	Modulus (Remainder)	5 % 3	2
**	Exponentiation (Power)	5 ** 3	125

3. Assignment Operators : Used to assign values to variables.

Operator	Example	Same as	Operator	Example	Same as
=	x = 5	x = 5	/=	x /= 3	x = x / 3
+=	x += 3	x = x + 3	%=	x %= 3	x = x % 3
-=	x -= 3	x = x - 3	//=	x //= 3	x = x // 3
*=	x *= 3	x = x * 3	**=	x **= 3	x = x ** 3

4. Comparison (Relational) Operators : Used to compare two values.

Operator	Description	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

5. Logical Operators : Used to combine conditions.

Operator	Description	Example	Result
and	True if both are True	(5 > 3) and (3 > 1)	True
or	True if any one is True	(5 < 3) or (3 > 1)	True
not	Reverse the result	not (5 < 3)	True

6. Membership Operators :

Used to test membership in a sequence.

Operator	Description	Example	Result
in	Present in	3 in [1,2,3,4]	True
not in	Not present in	5 not in [1,2,3]	True

7. Identity Operators :

Used to compare memory locations of objects.

Operator	Description	Example	Result
is	True if same object	a is b	True/False
is not	True if not same object	a is not b	True/False

8. Bitwise Operators :

Work with binary representation of integers.

Operator	Description	Example
&	Bitwise AND	5 & 3
|	Bitwise OR	5 | 3
^	Bitwise XOR	5 ^ 3
~	Bitwise NOT	~5
<<	Left Shift	5 << 1
>>	Right Shift	5 >> 1

Precedence and Associativity

When an expression contains multiple operators, Python evaluates it according to a fixed order of precedence.

If two operators have the same precedence, the associativity rule decides the order of evaluation.

1. Operator Precedence in Python

Precedence (High to Low)	Operator	Description	Example	Associativity
1	( )	Parentheses	(2 + 3) * 4	–
2	**	Exponentiation (Power)	2 ** 3	Right to Left
3	+x -x ~x	Unary plus, Unary minus, Bitwise NOT	-5 , +5 , ~5	Right to Left
4	* / // %	Multiplication, Division, Floor Division, Modulus	10 * 2 , 10 % 3	Left to Right
5	+ -	Addition, Subtraction	10 + 5 , 10 - 5	Left to Right
6	<< >>	Left shift, Right shift	5 << 1	Left to Right
7	&	Bitwise AND	5 & 3	Left to Right
8	^	Bitwise XOR	5 ^ 3	Left to Right
9	|	Bitwise OR	5 | 3	Left to Right
10	< <= > >=	Relational operators	5 < 3	Left to Right
11	== !=	Equality operators	5 == 3	Left to Right
12	not	Logical NOT	not True	Right to Left
13	and	Logical AND	True and False	Left to Right
	or	Logical OR	True or False	Left to Right

2. Associativity

If operators with the same precedence appear in an expression, the associativity rule decides in which direction they are evaluated.

- **Left to Right**
  Evaluation starts from the leftmost operator and moves to the right.
- **Right to Left**
  Evaluation starts from the rightmost operator and moves to the left.

3. Examples

- a) 12 + 5 * 2
  → Multiplication has higher precedence than addition.
  → So, 5 * 2 is evaluated first.
  = 12 + 10 = 22
- b) (12 + 5) * 2
  → Parentheses have highest precedence.
  → So, 12 + 5 is evaluated first.
  = 17 * 2 = 34
- c) 2 ** 3 ** 2
  → ** is right to left associative.
  → So, 3 ** 2 is evaluated first.
  = 2 ** 9 = 512
- d) 20 - 6 - 4
  → - has same precedence and is left to right.
  → (20 - 6) - 4 = 14 - 4 = 10
- e) True or False and False
  → and has higher precedence than or.
  → So, False and False is evaluated first.
  = True or False = True
- f) not True and False
  → not has higher precedence than and.
  → not True is evaluated first.
  = False and False = False

Summary :

1. Operators are evaluated according to precedence (High to Low).
2. If operators have the same precedence, associativity (Left to Right / Right to Left) determines the order.


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;
}

Data Types in Python

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

1. Built-in Data Types :

Data Type	Description	Example	Value	Type (in Python)
int (Integer)	Represents whole numbers (positive, negative or zero)	x = 10 y = -25	10 -25	<class 'int'>
float (Floating point)	Represents decimal numbers (numbers with fractional part)	pi = 3.14 rate = -0.75	3.14 -0.75	<class 'float'>
complex (Complex number)	Represents complex numbers in the form a + bj	z = 3 + 4j	(3+4j)	<class 'complex'>
bool (Boolean)	Represents truth values True or False	a = True b = False	True False	<class 'bool'>
str (String)	Represents sequence of characters (text)	name = "Yunus" msg = 'Hello'	"Yunus" 'Hello'	<class 'str'>
list (List)	Represents ordered collection of items (can be of different types)	L = [10, 20, 'abc', 3.5]	[10, 20, 'abc', 3.5]	<class 'list'>
tuple (Tuple)	Represents ordered collection of items (immutable)	T = (10, 20, 'abc')	(10, 20, 'abc')	<class 'tuple'>
set (Set)	Represents unordered collection of unique items	S = {10, 20, 30}	{10, 20, 30}	<class 'set'>
dict (Dictionary)	Represents collection of key-value pairs (unordered)	D = {'id': 1, 'name': 'Ali'}	{'id': 1, 'name': 'Ali'}	<class 'dict'>

2. Type Checking :

- We can check the data type of any value using the type() function.

Example : type(10) → <class 'int'> type(3.14) → <class 'float'> type("Yunus") → <class 'str'>

type(True) → <class 'bool'> type([1,2,3]) → <class 'list'> type({'a': 1}) → <class 'dict'>

Note : Python is a dynamically typed language, so we do not need to declare the type of a variable explicitly.

Indentation

Indentation refers to the spaces at the beginning of a line. In Python, indentation is very important. It is used to define a block of code.

- Python uses indentation to indicate a block of code.
- All statements within the same block must have the same indentation level.
- Unlike other languages that use { }, Python uses indentation.

Example :

Correct Indentation

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is 5 or less")
```

→ Here, print() statements are indented inside the if and else blocks.

Incorrect Indentation

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is 5 or less")
```

→ This will give an IndentationError because the indentation is incorrect.

Note :

- Use consistent indentation (usually 4 spaces) throughout the program.
- Do not mix tabs and spaces.

Comments

Comments are used to explain the code. They are ignored by the Python interpreter and are used to make the code more readable.

1. Single-line Comments

In Python, single-line comments start with the # symbol.

Example :

```
# This is a single-line comment
x = 10    # x stores the value 10
print(x)  # This will print the value of x
```

2. Multi-line Comments

Python does not have a special syntax for multi-line comments like /* ... */ in other languages. We can use triple quotes (''' or """) to write multi-line comments.

Example :

```
'''
This is a
multi-line comment.
It can span
across multiple lines.
'''
x = 25
print(x)
```

→ The text inside triple quotes is ignored by the interpreter.

Note : Comments are useful for documentation and understanding your code, especially in large programs.

Reading Input, Printing Output

1. Reading Input

In Python, we use the input() function to read data from the user.

```
Syntax :   variable = input(prompt)
```

- input() always reads the data as a string.
- We can convert it to other data types using type conversion functions like int(), float() etc.

Example 1 : Reading a string input

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

Sample Run :

```
Enter your name: Yunus
Hello, Yunus!
```

Example 2 : Reading integer input

```
num = int(input("Enter a number: "))
print("You entered:", num)
```

Sample Run :

```
Enter a number: 25
You entered: 25
```

Example 3 : Reading float input

```
price = float(input("Enter price: "))
print("Price is:", price)
```

Sample Run :

```
Enter price: 99.50
Price is: 99.5
```

Note : The input() function always returns a string. Use int(), float(), etc. to convert it to the required data type.

2. Printing Output

In Python, we use the print() function to display output on the screen.

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

- sep is used to separate values (default is space).
- end is used to end the line (default is newline).

Example 1 : Printing a single value

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

Output :

```
Hello, Python!
```

Example 2 : Printing multiple values

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

Output :

```
Name: Yunus Age: 20
```

Example 3 : Using sep parameter

```
print(10, 20, 30, sep=' - ')
```

Output :

```
10 - 20 - 30
```

Example 4 : Using end parameter

```
print("Hello", end=' ')
print("Python!")
```

Output :

```
Hello Python!
```

Note : The print() function is used to show results, messages or any output to the user.

Type Conversions

Type conversion (or type casting) is the process of converting a value from one data type to another.
In Python, we can convert data types using built-in functions.

1. Implicit Type Conversion (Automatic)

Python automatically converts a smaller data type to a larger data type to avoid data loss.

Example :

```
a = 10        # int
b = 3.5       # float
c = a + b     # int is converted to float
print(c)                    Output : 13.5
```

2. Explicit Type Conversion (Manual)

We can convert one data type to another using built-in functions.

Common type conversion functions :

Function	Converts to	Example	Result
int( )	Integer	int(3.9)	3
float( )	Float	float(10)	10.0
str( )	String	str(25)	'25'
bool( )	Boolean	bool(0)	False

3. Examples of Type Conversion

a) int( ) – to convert to integer

```
x = 7.89
y = int(x)
print(y)
```

Output : 7 # decimal part is removed

b) float( ) – to convert to float

```
x = '25.6'
y = float(x)
print(y)
```

Output : 25.6 # string is converted to float

c) str( ) – to convert to string

```
x = 100
y = str(x)
print(y)
```

Output : '100' # number is converted to string

d) bool( ) – to convert to boolean

```
x = 0
y = bool(x)
print(y)
```

Output : False # 0 is False in boolean

4. Type Conversion in Expressions

```
a = '15'        # string
b = 5           # integer
c = int(a) + b  # '15' is converted to 15
print(c)        # 15 + 5 = 20      Output : 20
```

Note :

- While converting, make sure the value is in a valid format.
- Invalid conversion will raise an error.

  Example :

  int('abc') will give a ValueError.

The type() Function and Is Operator

1. The type() Function

The type() function is used to find the data type of a value or variable. It returns the class type of the object.

Syntax : `type(object)`

Example :

Code	Output	Explanation
`x = 10` `print(type(x))`	`<class 'int'>`	x is an integer
`y = 3.14` `print(type(y))`	`<class 'float'>`	y is a float
`name = "Yunus"` `print(type(name))`	`<class 'str'>`	name is a string
`is_valid = True` `print(type(is_valid))`	`<class 'bool'>`	is_valid is boolean
`lst = [1, 2, 3]` `print(type(lst))`	`<class 'list'>`	lst is a list
`tpl = (1, 2, 3)` `print(type(tpl))`	`<class 'tuple'>`	tpl is a tuple
`d = {'a': 1, 'b': 2}` `print(type(d))`	`<class 'dict'>`	d is a dictionary

Note : type() tells us what type (class) the object belongs to. It does not check the value, only the data type.

2. Is Operator

The is operator is used to check whether two variables refer to the same object in memory.

Syntax : `variable1 is variable2`

Example 1 :

```
a = 100
b = 100
print(a is b)      # Output : True
```

→ Both a and b refer to the same object with value 100 (small integers are cached).

Example 2 :

```
x = [1, 2, 3]
y = x
print(x is y)      # Output : True
```

→ y refers to the same list object as x.

Example 3 :

```
p = [1, 2, 3]
q = [1, 2, 3]
print(p is q)      # Output : False
```

→ p and q have the same values but are different objects in memory.

Key Points :

- 'is' checks identity (same object in memory).
- It is not used to compare values.
- Use == to compare values.

Note : Use 'is' carefully. For most comparisons of values, use ==.

Dynamic and Strongly Typed Language

1. Dynamic Language

A dynamic language is one in which the data type of a variable is checked and decided at runtime.

- We do not need to declare the type of a variable.
- The type can change during program execution.
- More flexible and easier to write code.

Example (Python) :

```
x = 10          # x is an integer
print(type(x))  # Output: <class 'int'>
x = "Yunus"     # now x is a string
print(type(x))  # Output: <class 'str'>
x = 25.5        # now x is a float
print(type(x))  # Output: <class 'float'>
```

Key Point :

- Type of variable is determined at runtime.
- Same variable can hold different types of values.
- Example languages: Python, JavaScript, Ruby, PHP.

2. Strongly Typed Language

A strongly typed language is one in which the data type of a variable is checked at compile time.

- We must declare the type of a variable.
- The type cannot be changed once declared.
- Less flexible but safer and reduces type-related errors.

Example (Java) :

```
int x = 10;
System.out.println(x);  // Output: 10

x = "Yunus";  // Error: Type mismatch
// (cannot assign String to int)
```

Key Point :

- Type of variable is determined at compile time.
- Once declared, a variable cannot hold a different type.
- Example languages: Java, C, C++, C#, Rust.

Note : Python is both dynamic and strongly typed.

- ➤ Dynamic – type is decided at runtime.
- ➤ Strongly Typed – operations are performed only on compatible types (e.g., int + int is allowed, int + "text" is not allowed).

Control Flow Statements: The if, if-else

Control flow statements are used to control the execution of a program based on certain conditions.

1. The if Statement

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

Syntax :

```
if condition :
    # block of code to be executed
      if condition is True
```

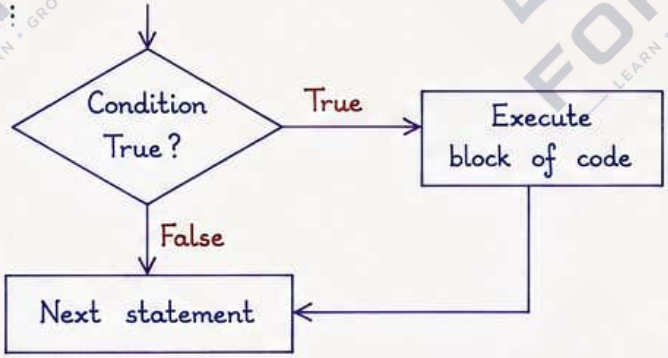
Example :

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

Output :

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

Flowchart :



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 the condition is False.

Syntax :

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

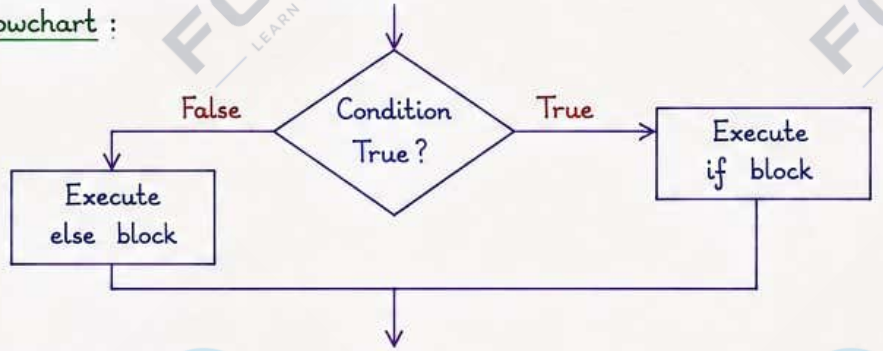
Example :

```
marks = 60
if marks >= 40 :
    print("You passed!")
else :
    print("You failed!")
```

Output :

```
You passed!
```

Flowchart :



Note :

- In the if statement, the else part is optional.
- Indentation is very important in Python.
- Code inside the if block runs only when the condition is True.
- In if-else, either the if block or the else block will be executed, not both.

if - elif - else Decision Control Statement

The if - elif - else statement is used to execute one block of code from multiple alternatives based on different conditions.

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
```

2. Example (Python) :

```
marks = int(input("Enter your marks (0-100): "))
if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 50:
    print("Grade: C")
elif marks >= 35:
    print("Grade: D")
else:
    print("Grade: Fail")
```

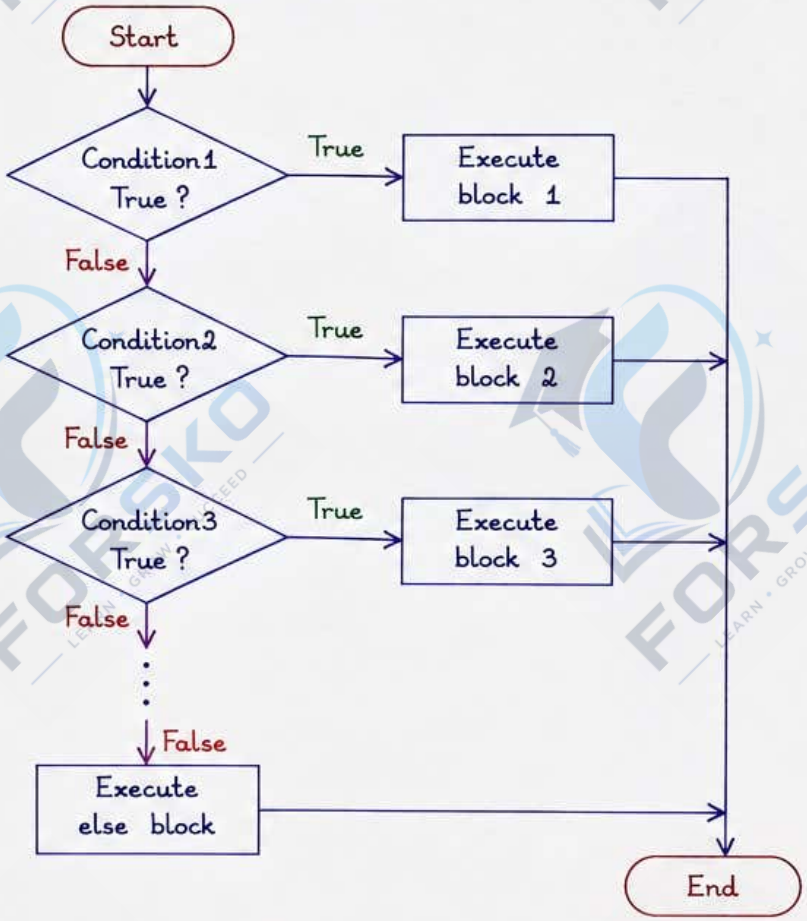
Sample Run :

```
Enter your marks (0-100): 82
Grade: B
```

```
Enter your marks (0-100): 56
Grade: C
```

```
Enter your marks (0-100): 28
Grade: Fail
```

3. Flowchart :



Note :

- The program checks conditions from top to bottom.
- As soon as one condition is True, its block is executed and the rest are skipped.
- If none of the conditions are True, the else block is executed.

Nested if Statement

A nested if statement is an if statement inside another if (or else) statement.
It is used to check multiple conditions (one condition inside another).

1. Syntax :

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

2. Example (Python) :

```
marks = int(input("Enter your marks (0-100): "))
if marks >= 50:
    if marks >= 75:
        print("Distinction")
    else :
        print("First Class")
else :
    if marks >= 35:
        print("Pass Class")
    else :
        print("Fail")
```

3. Sample Runs :

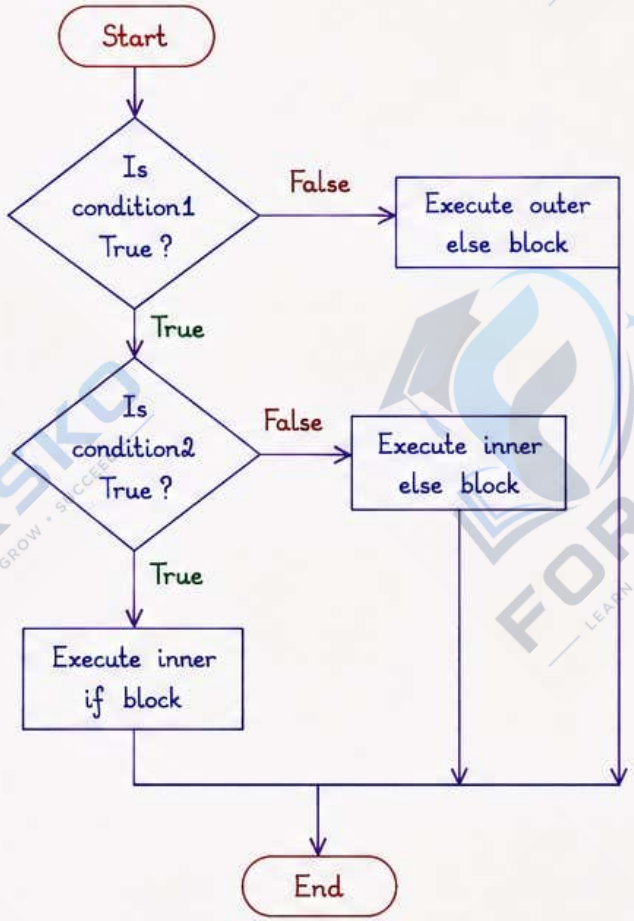
```
Enter your marks (0-100): 85
Distinction
```

```
Enter your marks (0-100): 60
First Class
```

```
Enter your marks (0-100): 40
Pass Class
```

```
Enter your marks (0-100): 20
Fail
```

4. Flowchart :



Note :

- The inner if-else is executed only when the outer if condition is True.
- If the outer if condition is False, the inner if is skipped and the outer else block is executed.
- Indentation is very important in nested if statements.

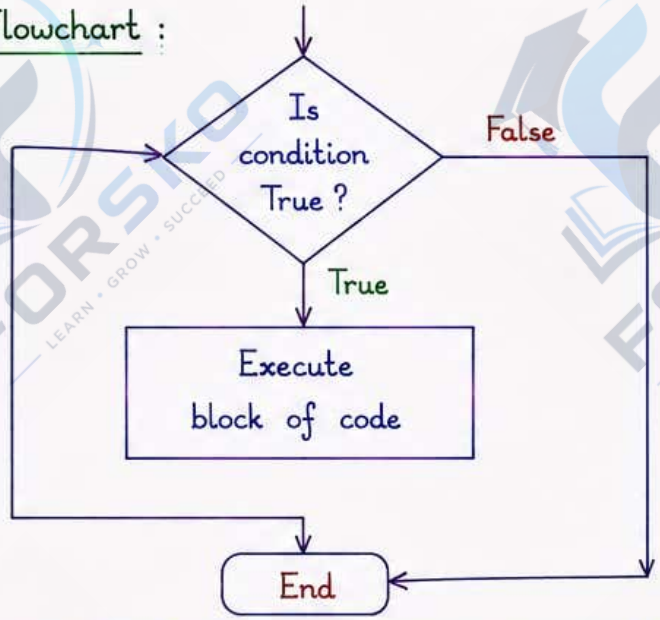
The while Loop

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

1. Syntax :

```
while condition :
    # block of code
    # (repeated while condition is True)
```

2. Flowchart :



3. Example (Python) :

```
i = 1                       # initialization
while i <= 5:               # condition
    print(i, end = " ") # block of code
    i = i + 1               # update
```

Output : 1 2 3 4 5

4. Example Explanation :

- Initially, i = 1.
- Check condition: i <= 5 → True → print 1 and i becomes 2.
- Again check: i <= 5 → True → print 2 and i becomes 3.
- This continues until i = 5.
- After printing 5, i becomes 6.
- Now, i <= 5 → False, so loop terminates.

Key Points :

- The block of code inside the while loop is executed zero or more times.
- If the condition is False at the beginning, the block will not execute even once.
- Make sure to update the condition inside the loop, otherwise it may cause an infinite loop.

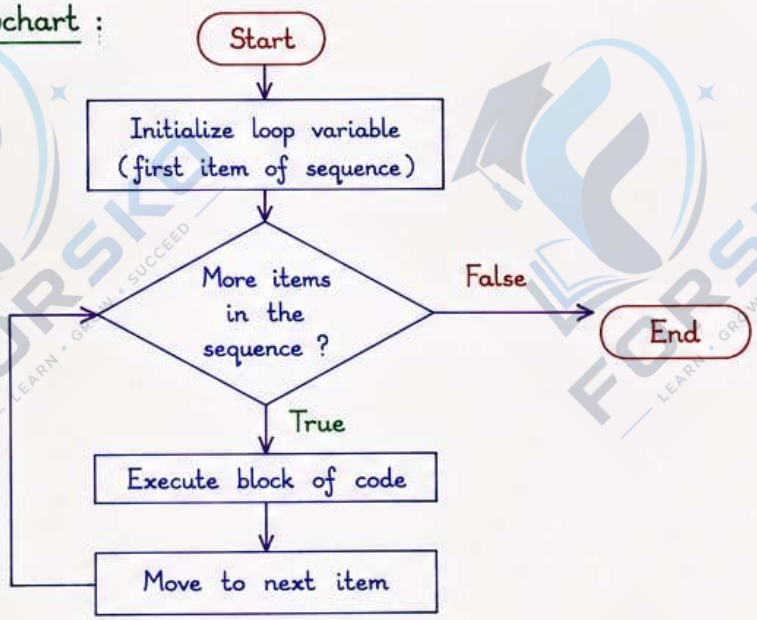
The for Loop

The for loop is used to iterate over a sequence (such as list, tuple, string, range etc.) and execute a block of code for each item.

1. Syntax :

```
for variable in sequence :
    # block of code
    # (repeated for each item in the sequence)
```

2. Flowchart :



3. Example (Python) :

```
fruits = ["Apple", "Banana", "Mango", "Orange"]   # sequence (list)
for fruit in fruits :                             # loop through each item
    print(fruit)                                  # block of code
```

Output :

```
Apple
Banana
Mango
Orange
```

4. Example using range() :

```
for i in range(1, 6):      # range from 1 to 5
    print(i, end = " ")    # print numbers on same line
```

Output :

```
1 2 3 4 5
```

5. Explanation :

- The loop variable takes the first item from the sequence.
- The block of code is executed.
- Then the loop variable takes the next item and the block is executed again.
- This continues until all items in the sequence are processed.
- When there are no more items, the loop ends.

Key Points :

- The for loop is used when the number of iterations is known (or we want to iterate over a sequence).
- It can be used with lists, tuples, strings, sets, dictionaries, and the range() function.
- It automatically takes each item one by one from the sequence.
- No need to manually update the loop variable.

The continue and break Statements.

The continue and break statements are used inside loops to change the normal flow of execution.

1. The continue Statement :

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

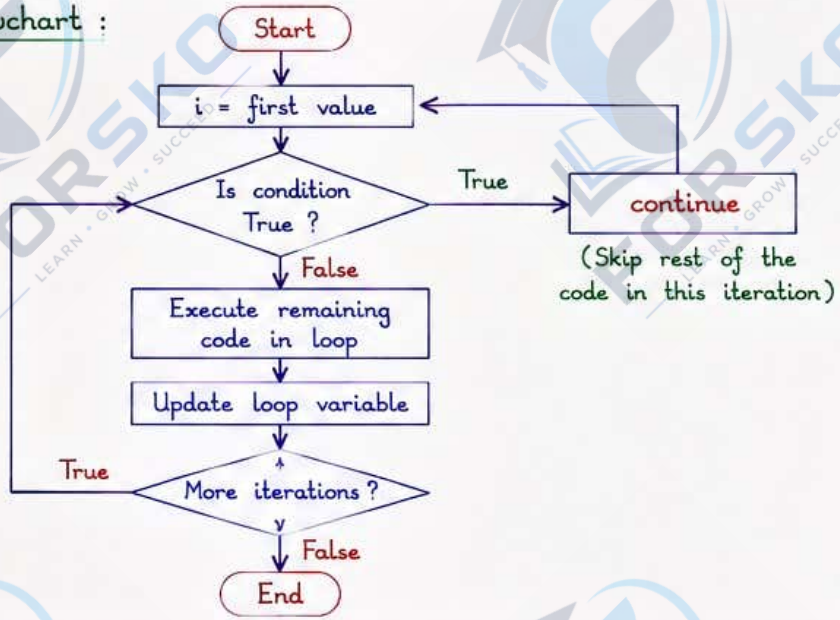
Example (Python) :

```
for i in range(1, 6):
    if i == 3:
        continue      # skip when i is 3
    print(i, end = " ")
```

Output :

```
1 2 4 5
```

Flowchart :



2. The break Statement :

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

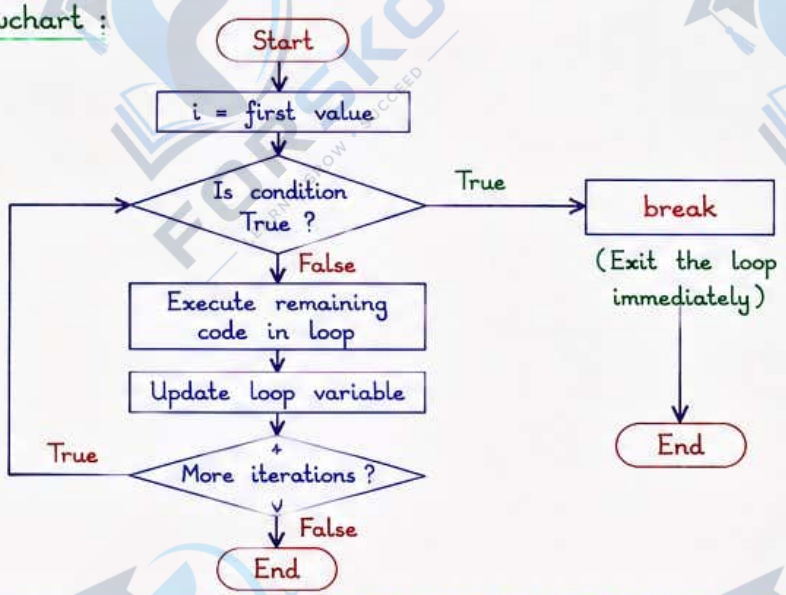
Example (Python) :

```
for i in range(1, 6):
    if i == 3:
        break      # exit the loop when i is 3
    print(i, end = " ")
print("Loop ended.")
```

Output :

```
1 2
Loop ended.
```

Flowchart :



Key Points :

- continue skips the rest of the code in the loop for the current iteration and moves to the next iteration.
- break terminates the loop completely.
- Both statements are used inside loops (for or while).
- They are useful for controlling the flow of the program.


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

Functions: Built-In Functions

Python provides a number of built-in functions that are always available for use.
They perform common tasks and make programs easier.

1. Common Built-In Functions :

Function	Description	Example	Result
print()	Displays the output on the screen	print("Hello, Python")	Hello, Python
type()	Returns the type of the object	type(25.5)	<class 'float'>
int()	Converts the value to integer	int("123")	123
float()	Converts the value to float	float("12.34")	12.34
str()	Converts the value to string	str(100)	'100'
len()	Returns the length of the object	len("Python")	6
input()	Takes input from the user	input("Enter name: ")	(User input)
max()	Returns the largest value	max(10, 25, 5)	25
min()	Returns the smallest value	min(10, 25, 5)	5
abs()	Returns the absolute value	abs(-7)	7
round()	Rounds the number to nearest integer	round(4.67)	5

2. Example Program Using Built-In Functions :

```
name = input("Enter your mame: ")                    # input()
age = int(input("Enter your age: "))                 # int()
print("Hello", name)                                 # print()
print("Next year, you will be", age + 1)             # + operator
print("Type of age:", type(age))                     # type()
print("Length of name:", len(name))                  # len()
```

Sample Run :

```
Enter your name: Yunus
Enter your age: 20
Hello Yunus
Next year, you will be 21
Type of age: <class 'int'>
Length of name: 5
```

3. Explanation :

- input() gets data from the user.
- int() converts the input to integer.
- print() displays output.
- type() returns the data type.
- len() gives the number of characters in the name.

Key Points :

- Built-in functions are predefined in Python.
- They save time and reduce the need to write complex code.
- Some of the most commonly used built-in functions are: print(), type(), len(), input(), int(), float(), max(), min(), abs(), round(), etc.

Commonly Used Modules

Python provides a large number of standard modules that extend its functionality.
Some of the most commonly used modules are listed below.

1. List of Commonly Used Modules :

Module	Purpose	Commonly Used Functions	Example
math	Mathematical functions and constants	sqrt(), pow(), sin(), cos(), pi, etc.	math.sqrt(16) → 4.0
random	Generate random numbers and choices	randint(), choice(), random(), shuffle(), etc.	random.randint(1,10) → e.g. 7
datetime	Work with dates and times	datetime(), date(), now(), strftime(), etc.	datetime.now() → 2024-05-20 10:30:00
time	Time related functions	time(), sleep(), ctime(), localtime(), etc.	time.sleep(2) → waits for 2 seconds
os	Interact with operating system	listdir(), mkdir(), remove(), path, etc.	os.listdir('.') → list of files
sys	System-specific parameters and functions	argv, exit(), path, version, etc.	sys.version → Python version
re	Regular expressions (pattern matching)	match(), search(), findall(), sub(), etc.	re.findall(r"\d+", "A1B2") → ["1", "2"]
json	Work with JSON data	dumps(), loads(), load(), dump(), etc.	json.dumps({"a":1}) → "{"a": 1}"
collections	Special container datatypes	Counter(), defalutdict(), namedtuple(), deque(), etc.	Counter('banana') → Counter({'a':3,'n':2,'b':1})

2. How to Use a Module :

- Import the entire module

```
import module_name                    # access with module_name.function()
```

- Import specific function(s)

```
from module_name import function1, function2    # use function directly
```

- Import module with an alias

```
import module_name as alias           # access with alias.function()
```

3. Examples :

(i) math module

```
import math
print(math.sqrt(25))                  # Output: 5.0
```

(ii) random module

```
from random import randint
print(randint(1, 100))                # Output: any number between 1 and 100
```

(iii) datetime module

```
from datetime import datetime
print(datetime.now())                 # Output: current date and time
```

(iv) os module

```
import os
print(os.getcwd())                    # Output: current working directory
```

Key Points :

- Modules provide pre-written code that can be used in our programs.
- Using modules saves time and effort.
- We can import entire modules or specific functions as per our requirement.
- Some modules are built-in (standard library) and some need to be installed using pip.

Function Definition and Calling the Function

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

1. Function Definition :

A function is defined using the keyword **def**, followed by the function name, parentheses (), and a colon :

```
def greet(name):                    # function definition
    print("Hello, " + name + "!")   # function body
                                      (indented block)
```

Explanation :

- **def** is the keyword used to define a function.
- **greet** is the function name.
- **name** is a parameter (input).
- The indented block is the body of the function.

2. Calling the Function :

A function is executed (run) only when it is **called**.

```
greet("Yunus")        # function call
```

Output :

```
Hello, Yunus!
```

3. Example Program :

```
# Function definition
def add(a, b):
    sum = a + b           # statement inside function
    print("Sum is:", sum) # print the result

# Calling the function
add(10, 20)       # function call
```

Output :

```
Sum is: 30
```

4. Explanation :

- add(a, b) is a function that takes two parameters a and b.
- When add(10, 20) is called, the values 10 and 20 are passed to a and b respectively.
- The function performs the addition and prints the result.

Key Points :

- A function must be defined before it is called.
- Parameters are the values received by the function.
- Arguments are the actual values passed while calling the function.
- A function can be called multiple times.

The return Statement and void Function

In Python, a function can return a value back to the caller using the **return** statement.
A function that does not return any value is called a **void function**.

1. The return Statement :

- The **return** statement is used to send a value from the function back to the caller.
- As soon as **return** is executed, the function terminates and control is transferred back to the caller.

```
def add(a, b):                  # function definition
    sum = a + b                 # calculate sum
    return sum                  # return the result
```

```
# Calling the function
result = add(15, 25)        # function call and store returned value
print("Sum:", result)       # Output: Sum: 40
```

2. void Function :

- A function that does not return any value is called a **void function**.
- It is used to perform some task without returning any result.

```
def greet(name):                        # void function definition
    print("Hello, " + name + "!")       # perform task
    # No return statement               # nothing is returned
```

```
# Calling the void function
greet("Yunus")                          # function call
print("Function call completed.")       # Execution continues here
```

Output :

```
Hello, Yunus!
Function call completed.
```

3. Example : Using both return and void functions

```
# Function with return
def square(n):
    return n * n

# Function without return (void)
def show(msg):
    print("Message:", msg)
```

```
# Main program
x = square(6)
print("Square is:", x)          # uses returned value

show("Welcome to Python!")      # void function call

print("Program ended.")
```

Output :

```
Square is: 36
Message: Welcome to Python!
Program ended.
```

Key Points :

- **return** sends a value from the function back to the caller.
- A function with **return** must return a value.
- **void function** does not return any value.
- A void function is used to perform actions (like printing, updating, etc.).
- Even after calling a void function, the program continues execution.

Scope and Lifetime of Variables

The scope of a variable defines the part of a program where the variable can be accessed.

The lifetime of a variable defines the period for which the variable exists in memory.

1. Scope of Variables :

Scope determines the visibility or accessibility of a variable in different parts of a program.

Type of Variable	Scope	Example

Local Variable	Declared inside a function or block.
	Can be accessed only within that function or block.
		`def fun():`
		`  x = 10  # local variable`
		`  print(x)`
		`fun()`
		`# x cannot be accessed here`

Global Variable	Declared outside all functions.
	Can be accessed from any function in the program.
		`y = 20  # global variable`
		`def fun():`
		`  print(y)  # accessible`
		`fun()`
		`print(y)`

Parameter Variable	Declared in the function definition.
	Acts as a local variable within that function.
		`def add(a, b):  # a and b are`
		`  c = a + b  # parameters`
		`  print(c)`
		`add(5, 6)`

Key Points :

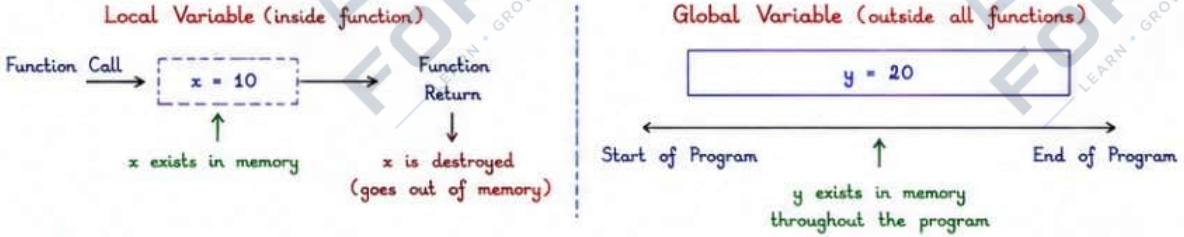
- Local variables have local scope.
- Global variables have global scope.
- Parameter variables have local scope (within the function).
- A variable can have only one scope at a time.

2. Lifetime of Variables :

Lifetime determines the time period from creation to destruction of a variable.

Type of Variable	Lifetime	Explanation
Local Variable	From the time the block or function is entered till it is exited.	Created when the function/block starts execution and destroyed when it exits.
Global Variable	From the start of the program till the end of the program.	Exists throughout the execution of the program.
Parameter Variable	From the time the function is called till it returns.	Created when the function is called and destroyed when the function returns.

3. Visualization (Memory Representation)



4. Example :

```
x = 100     # global variable
def show():
    y = 20    # local variable
    print("Inside function, y =", y)  # accessible here
show()
print("Outside function, x =", x)  # accessible
# print(y)  # Error: y is not accessible here
```

```
# x is global, exists throughout the program.
# y is local, exists only during function call.
# After function ends, y is destroyed.
# x can be used anywhere in the program.
```

Default Parameters

Definition :

Default parameters are the values assigned to parameters in a function definition. If no value is passed during function call, the default value is used.

Syntax :

```
def function_name(parameter = default_value):
    statement(s)
```

Example 1 :

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

# Function calls
greet("Yunus")                  # uses default msg
greet("Yunus", "Good Morning")  # uses provided msg

# Output
Hello, Yunus!
Good Morning, Yunus!
```

Example 2 :

```
def add(a, b = 10, c = 20):
    sum = a + b + c
    print("Sum =", sum)

# Function calls
add(5)          # b and c are default
add(5, 15)      # c is default
add(5, 15, 25)  # no default

# Output
Sum = 35
Sum = 45
Sum = 45
```

Important Points :

- Default parameters make functions more flexible and easy to use.
- Default values are used only when the argument is not provided.
- Default parameters must be at the end of tha parameter list.

Rules :

1. A parameter with a default value cannot precede a parameter without a default value.
2. Default values are evaluated only once, when the function is defined.
3. They are helpful in reducing the number of arguments in a function call.

Example (Invalid) :

```
def invalid(a = 10, b):   # ✗ Wrong
                          # SyntaxError
```

Example (Valid) :

```
def valid(a, b = 10):     # ✓ Correct
```

Real Life Analogy :

Think of a function like ordering food. Some items (like water) are default (automatically provided), but you can also choose extra items if you want.

Keyword Arguments

Definition :

Keyword arguments are arguments passed to a function by explicitly specifying the parameter name along with its value. This allows the order of arguments to be different from the order of parameters in the function definition.

Why use Keyword Arguments ?

- Improves readability of the code.
- You don't need to remember the exact order of parameters.
- Makes the code more clear and self-explanatory.
- Reduces errors, especially when a function has many parameters.

Syntax :

```
def function_name(param1, param2, ..., paramN):
    statement(s)
```

```
function_name(arg_name1 = value1,
              arg_name2 = value2, ...)
```

Example 1 :

```
def student(name, age, course):
    print(f"Name   : {name}")
    print(f"Age    : {age}")
    print(f"Course : {course}")

# Using keyword arguments (different order)
student(course = "Python", name = "Yunus", age = 20)

# Output
Name   : Yunus
Age    : 20
Course : Python
```

Example 2 :

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

# Using keyword arguments
result = add(c = 30, a = 10, b = 20)

# Output
result = 60
```

Example 3 : (Mixing positional and keyword arguments)

```
def greet(greeting, name, msg = "Have a nice day!"):
    print(f"{greeting}, {name}! {msg}")

# Positional argument for 'greeting',
# keyword arguments for others
greet("Hello", name = "Yunus", msg = "Good morning!")

# Output
Hello, Yunus! Good morning!
```

Important Points :

- Keyword arguments must come after positional arguments.
- You cannot provide a parameter more than once.
- All parameters can be passed as keyword arguments.

Example (Invalid) :

```
def display(a, b):
    print(a, b)

# Invalid: a is given twice
display(10, a = 20)        ✗ Wrong
# Correct
display(10, b = 20)        ✓ Correct
```

*args and **kwargs

1. *args (Non-Keyword Arbitrary Arguments)

*args allows you to pass a variable number of non-keyword (positional) arguments to a function. These arguments are received as a tuple inside the function.

Syntax :

```
def function_name(*args):
    statement(s)
```

Example :

```
def add(*args):
    total = 0
    for num in args:
        total += num
    return total

# Function call
add(1, 2)              # 3
add(1, 2, 3, 4)        # 10
add(5, 10, 15, 20, 25) # 75
```

2. **kwargs (Keyword Arbitrary Arguments)

**kwargs allows you to pass a variable number of keyword arguments to a function. These arguments are received as a dictionary inside the function.

Syntax :

```
def function_name(**kwargs):
    statement(s)
```

Example :

```
def display_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} : {value}")
# Function call
display_details(name="Yunus", age=20, city="Pune")
# Output
name    : Yunus
age     : 20
city    : Pune
```

3. Using *args and **kwargs together

Syntax :

```
def function_name(*args, **kwargs):
    statement(s)
```

Example :

```
def student_info(*args, **kwargs):
    print("Subjects:", args)
    print("Other Info:", kwargs)
# Function call
student_info("Maths", "Physics", "Chemistry",
             name="Yunus", age=20, city="Pune")
# Output
Subjects: ('Maths', 'Physics', 'Chemistry')
Other Info: {'name': 'Yunus', 'age': 20, 'city': 'Pune'}
```

4. Key Points

- *args collects extra positional arguments as a tuple.
- **kwargs collects extra keyword arguments as a dictionary.
- *args must come before **kwargs in the function definition.
- They make functions more flexible and reusable.

Quick Comparison

	*args	**kwargs
Accepts	Extra positional arguments	Extra keyword arguments
Type	Tuple	Dictionary
Access	By index	By key
Example	func(1, 2, 3)	func(a=1, b=2)

Command Line Arguments

★ Definition :

Command line arguments are the values passed to a Python program when it is run from the command prompt or terminal. These arguments are read from the `sys.argv` list.

★ How it works ?

- Python stores all command line arguments in a list called `sys.argv`.
- `sys.argv[0]` → contains the name of the script itself.
- `sys.argv[1]`, `sys.argv[2]`, ... → contain the actual arguments passed by the user.

★ Syntax :

```
python script_name.py arg1 arg2 ... argN
```

★ Example :

```
python add.py 10 20
# Here, 10 and 20 are
  command line arguments.
```

★ Example Program : (add.py)

```
import sys
# sys.argv is the list of command line arguments
print("Total number of arguments:", len(sys.argv))
print("Script name:", sys.argv[0])
print("First argument:", sys.argv[1])
print("Second argument:", sys.argv[2])

# Convert arguments to integers and add
a = int(sys.argv[1])
b = int(sys.argv[2])
c = a + b
print("Sum =", c)
```

★ Output :

```
$ python add.py 10 20
Total number of arguments: 3
Script name: add.py
First argument: 10
Second argument: 20
Sum = 30
```

★ Important Points :

- The number of arguments = `len(sys.argv)`
- Always check the number of arguments before using them.
- Convert arguments to required data types (`int`, `float`, etc.) before using.

★ Check number of arguments :

```
import sys
if len(sys.argv) < 3:
    print("Usage: python add.py num1 num2")
    sys.exit()
# Program continues if enough arguments are provided
```

★ Notes :

- `sys.argv` is available in the `sys` module.
- First argument (`sys.argv[0]`) is always the script name.
- Used in automation, file processing, configuration, and many real-life applications.


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;
}

Strings: Creating and Storing Strings

★ 1. What is a String?

A string is a sequence of characters enclosed in single quotes, double quotes or triple quotes.

★ 2. Creating Strings

Strings can be created by enclosing characters in quotes.

```
Single quotes        'Hello'       # using single quotes
Double quotes        "Hello"       # using double quotes
Triple single quotes '''Hello'''   # using triple single quotes
Triple double quotes """Hello"""   # using triple double quotes
```

3. Storing Strings in Variables

A string can be stored in a variable just like any other data.

```
name = "Yunus"                  # double quotes
city = 'Pune'                   # single quotes
message = "Welcome to Python"   # string stored in variables
```

Note :
Python treats single and double quoted strings the same way.

4. Examples

```
# Different ways to create strings
s1 = 'Python'
s2 = "Programming"
s3 = '''Hello World'''
s4 = """This is a
        multi-line
        string"""

# Strings stored in variables
first_name = "Yunus"
last_name = 'Shaikh'
full_name = first_name + " " + last_name

# Accessing and printing strings
print(s1)
print(s2)
print(s3)
print(s4)
print(full_name)   # Output: Yunus Shaikh
```

5. Escape Sequences in Strings

Special characters can be included in strings using escape sequences.

Escape Sequence	Meaning	Example
\n	New line	"Hello\nWorld"
\t	Tab	"Name:\tYunus"
\\	Backslash	"C:\\Python"
\'	Single quote	'It\'s easy'
\"	Double quote	"She said,\"Hi\""

6. Multi-line Strings

Triple quotes are used to create multi-line strings.

```
info = """Python is a powerful language.
          It is easy to learn.
          It is widely used in industry."""
```

Output :

```
Python is a powerful language.
It is easy to learn.
It is widely used in industry.
```

Basic String Operations

1. Concatenation (Joining)

Strings can be joined together using the + operator.

Example :

```
s1 = "Hello"
s2 = "World"
result = s1 + " " + s2      # Hello World
```

2. Repetition

A string can be repeated using the * operator.

Example :

```
s = "Hi"
result = s * 3      # HiHiHi
```

3. Length

The built-in function len() returns the number of characters in a string.

Example :

```
s = "Python"
length = len(s)      # 6
```

> Note : In Python, strings are immutable. This means their contents cannot be changed after creation.

4. Indexing

Individual characters in a string can be accessed using their index (position).

Example :

```
    s = "Python"
Index :  0   1   2   3   4   5
Char  :  P   y   t   h   o   n

s[0]    # 'P'
s[3]    # 'h'
s[-1]   # 'n'  (negative index starts from end)
```

5. Slicing

A part of a string can be extracted using slicing [start:end].

Example :

```
s = "Python"
s[0:3]    # 'Pyt'   (start to end-1)
s[2:5]    # 'tho'
s[:4]     # 'Pyth'  (start to index 4)
s[3:]     # 'hon'   (from index 3 to end)
s[-3:]    # 'hon'   (last 3 characters)
```

6. Membership

We can check if a substring exists in a string using in and not in operators.

Example :

```
s = "Python Programming"
"Python" in s      # True
"Java" in s        # False
```

7. Common String Methods

Method	Purpose	Example
upper()	Converts to uppercase	"hello".upper() # 'HELLO'
lower()	Converts to lowercase	"HELLO".lower() # 'hello'
title()	Converts to title case	"hello world".title() # 'Hello World'
strip()	Removes leading and trailing spaces	" hello ".strip() # 'hello'
replace(a,b)	Replaces all a with b	"one two one".replace("one","1") # '1 two 1'
split(sep)	Splits string into a list using separator sep	"a,b,c".split(",") # ['a', 'b', 'c']
find(sub)	Returns index of first occurrence of sub	"hello".find("e") # 1 "hello".find("x") # -1
count(sub)	Counts occurrences of sub in string	"banana".count("a") # 3

8. Comparison

Strings can be compared using comparison operators.

```
"abc" == "abc"   # True     "abc" != "abd"   # True     "abc" < "abd"   # True (lexicographical order)
```

9. Empty String

An empty string contains no characters.

```
s = ""      len(s)      # 0
```

Accessing Characters in String by Index Number

1. Introduction

Each character in a string has a unique index (position) number.

We can access any character of a string by using its index number.

In Python, indexing starts from 0.

2. Positive Indexing (Left to Right)

The index of the first character is 0, the next is 1, then 2, and so on.

Example :

```
s = "Python"
```

Index	0	1	2	3	4	5
String	P	y	t	h	o	n

Accessing Characters :

```
s[0]  →  'P'   # First character
s[1]  →  'y'   # Second character
s[2]  →  't'   # Third character
s[3]  →  'h'   # Fourth character
s[4]  →  'o'   # Fifth character
s[5]  →  'n'   # Sixth character
```

Note :
If we try to access an index that is not available in the string, Python will give an IndexError.

3. Negative Indexing (Right to Left)

We can also access characters from the end of the string using negative index numbers.

The last character has index -1, the one before it has index -2, and so on.

Example :

```
s = "Python"
```

Index	0	1	2	3	4	5
String	P	y	t	h	o	n
Neg. Index	-6	-5	-4	-3	-2	-1

Accessing Characters :

```
s[-1]  →  'n'   # Last character
s[-2]  →  'o'   # Second last character
s[-3]  →  'h'   # Third last character
s[-4]  →  't'   # Fourth last character
s[-5]  →  'y'   # Fifth last character
s[-6]  →  'P'   # Sixth last character
```

5. General Syntax

```
string_name[index]
```

→ string_name → Name of the string

→ index → Index number (int)

4. Examples

Example 1 :

```
s = "Hello"
print(s[0])     # Output: H
print(s[1])     # Output: e
print(s[4])     # Output: o
print(s[-1])    # Output: o
print(s[-2])    # Output: l
print(s[-5])    # Output: H
```

Example 2 :

```
s = "Programming"
print(s[0])     # P
print(s[5])     # a
print(s[-1])    # g
print(s[-4])    # m
```

Important Points :

- Indexing starts from 0.
- Positive index → Left to Right
- Negative index → Right to Left
- Range of index for a string of length n is from -(n) to (n-1)
- Strings are immutable, so we can only access, not change characters.

String Slicing and Joining

1. String Slicing

Slicing allows us to extract a part (substring) from a string using the slice operator `[start : end : step]`.

- start → starting index (inclusive) [default is 0]
- end → ending index (exclusive) [default is length of string]
- step → interval between characters [default is 1]

Example :

```
s = "↑ Python Programming"
```

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
String	P	y	t	h	o	n	' '	P	r	o	g	r	a	m	m	i	n	g
Neg. Index	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Common Slicing Examples :

- `s[0:6]` → `'Python'` # from index 0 to 5
- `s[7:18]` → `'Programming'` # from index 7 to 17
- `s[:6]` → `'Python'` # from start to index 5
- `s[7:]` → `'Programming'` # from index 7 to end
- `s[:]` → `'Python Programming'` # whole string
- `s[::2]` → `'Ptorgamn'` # every 2nd character
- `s[1:10:2]` → `'yhogr'` # from index 1 to 9, step 2
- `s[::-1]` → `'gnimmargorP nohtyP'` # reverse string
- `s[-10:-1]` → `'rogramming'` # negative indexing
- `s[-6:]` → `'ramming'` # last 6 characters

Note :

- The character at index 'end' is not included in the result.
- If start is greater than end (with positive step), result is empty.
- Negative step is used for reverse slicing.

2. String Joining

Joining is the process of combining multiple strings into a single string using a separator.

Syntax :

```
separator.join(iterable)
```
→ returns a new string

- separator → the string placed between elements
- iterable → any sequence (list, tuple, etc.) of strings

Example 1 :

```
words = ["Python", "is", "fun"]
result = " ".join(words)        # space separator
print(result)                   # Output: Python is fun
```

Example 2 :

```
",".join(words)                 # Output: Python,is,fun
"-".join(words)                 # Output: Python-is-fun
"|".join(["A","B","C","D"])     # Output: A|B|C|D
```

Important Points :

- The elements of iterable must be strings, otherwise TypeError occurs.
- join( ) is more efficient than using + operator in a loop.
- join( ) does not add separator at the end of the string.

Example Program :

```
fruits = ["Apple", "Banana", "Cherry"]
print(", ".join(fruits))        # Output: Apple, Banana, Cherry
print(" & ".join(fruits))       # Output: Apple & Banana & Cherry
```