Birla Vishvakarma Mahavidyalaya (BVM) Engineering College

An Autonomous Institution

Computer Workshop – II (103SB)

(LTP = 0-0-4)

Unit 6: Introduction of Python

❖ Introduction to Python Language

Python is a high-level, interpreted programming language known for its simplicity and versatility. Created by Guido van Rossum and first released in 1991, Python emphasizes readability and ease of use, making it an excellent choice for both beginners and experienced developers. Python's syntax allows programmers to express concepts in fewer lines of code compared to other languages, enhancing productivity and reducing the likelihood of errors.

Python is an open-source language, supported by a vibrant community of developers who continually contribute to its growth. Its design philosophy prioritizes code readability and simplicity, making it a popular choice for a wide range of applications.

➢ Advantages of Python

1. **Easy to Learn and Use**: Python's straightforward syntax mimics natural language, making it accessible to beginners.
2. **Versatility**: It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.
3. **Extensive Libraries**: Python has a rich standard library and thousands of third-party modules for tasks ranging from web development to data analysis.
4. **Platform Independence**: Python programs can run on various operating systems without requiring modification.
5. **Active Community Support**: The extensive Python community provides ample resources, tutorials, and forums for troubleshooting and learning.
6. **Integration Capabilities**: Python can easily integrate with other languages and technologies, such as C, C++, and Java, and supports APIs for seamless interaction.
7. **Automation and Scripting**: Python excels in automating repetitive tasks and building robust scripts.

➢ Applications of Python

1. **Web Development**: Frameworks like Django, Flask, and Pyramid allow developers to build scalable and secure web applications.
2. **Data Science and Analytics**: Python is a dominant language for data analysis, visualization, and machine learning, with libraries like Pandas, NumPy, and Scikit-learn.
3. **Artificial Intelligence and Machine Learning**: Python's flexibility and powerful libraries, such as TensorFlow and PyTorch, make it ideal for AI projects.
4. **Scientific Computing**: Researchers and scientists use Python for simulations, computational tasks, and numerical analysis with tools like SciPy and Matplotlib.
5. **Game Development**: Libraries such as Pygame allow developers to create games and interactive applications.
6. **Automation and Scripting**: Python is widely used for automating system tasks, data scraping, and test scripting.
7. **Embedded Systems**: Python can interface with hardware components and is used in Internet of Things (IoT) projects.
8. **Desktop Application Development**: GUI frameworks like Tkinter, PyQt, and Kivy enable the development of cross-platform desktop applications.

Python's adaptability and efficiency continue to make it one of the most sought-after programming languages across diverse industries, shaping the future of technology and innovation.

❖ Local Environment Setup

To start working with Python, you need to set up a local development environment. Follow these steps:

1. **Download and Install Python**:
   - Visit the official Python website and download the latest version suitable for your operating system.
   - Run the installer and ensure you select the option to add Python to the system PATH.
2. **Verify Installation**:
   - Open a terminal or command prompt.
   - Type **python --version** or **python3 --version** to confirm Python is installed.
3. **Set Up an Integrated Development Environment (IDE)**:
   - Popular IDEs include PyCharm, Visual Studio Code, and Jupyter Notebook.
   - Install your preferred IDE and configure it for Python development.

4. **Install Required Libraries**:
    - Use **pip**, Python's package manager, to install libraries.

      For example:

```
pip install numpy pandas matplotlib
```

5. **Test Your Setup**:
    - Create a file named hello.py with the following content:

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

    - Run the file in the terminal using:

```
python hello.py
```

For further details, refer to the official Python documentation.

❖ Basic Syntax of Python

Python is a high-level, interpreted programming language known for its simplicity and readability. It uses indentation to define code blocks, making it distinct from many other programming languages.

➢ Key Features of Python Syntax :-

1. **Case Sensitivity:** Python is case-sensitive. For example, Variable and variable are two different identifiers.

2. **Indentation:** Python uses indentation to define blocks of code. Incorrect indentation results in an error.

```
if True:
    print("This is indented correctly.")
```

3. **Comments:**
    - Single-line comments begin with #.
    - Multi-line comments use triple quotes (''' or """).

```
# This is a single-line comment
"""
This is a multi-line comment
spanning multiple lines.
"""
```

4. **Variables:** Variables in Python are dynamically typed and do not require explicit declaration.

```
x = 10  # Integer
y = 3.14  # Float
name = "Python"  # String
```

5. **Statements:** Python typically executes one statement per line. To write multiple statements on a single line, use a semicolon (**;**).

```
print("Hello"); print("World")
```

6. **Keywords:** Python has a set of reserved keywords that cannot be used as variable names (e.g., if, else, while, import).

```
import keyword
print(keyword.kwlist)  # Displays all keywords
```

7. **Input and Output:**
   - Input from the user: input()
   - Display output: print()

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

8. **Basic Data Types:** int, float, str, bool, list, tuple, dict, and set are commonly used.

```
age = 25  # int
pi = 3.14159  # float
is_valid = True  # bool
colors = ["red", "green", "blue"]  # list
```

➢ Exercise 1: Display a Message

Problem Statement: Write a Python program to display the message "Hello, Python!".

Solution:

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

➢ Exercise 2: Input Name and Age

Problem Statement: Create a program that asks the user for their name and age, then prints a message using the input.

Solution:

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print(f"Hello, {name}! You are {age} years old.")
```

➢ Exercise 3: Use of Multi-line Comments

Problem Statement: Write a Python program to demonstrate the use of multi-line comments.

Solution:

```
"""
This program demonstrates the use of multi-line comments.
Author: Your Name
Date: Today's Date
"""
print("Multi-line comments are useful for documentation.")
```

❖ Variable Types in Python

In Python, variables are used to store data, and their type is determined automatically based on the value assigned. Python supports several built-in data types that can be broadly categorized into the following:

1. Numeric Types

- **int**: Represents integers, e.g., 10, -5
- **float**: Represents decimal numbers, e.g., 3.14, -0.01
- **complex**: Represents complex numbers, e.g., 3 + 4j

```
x = 10  # int
y = 3.14  # float
z = 2 + 3j  # complex
```

2. Sequence Types

- **str (String)**: A sequence of characters enclosed in quotes (single or double).

```
name = "Python"
```

- **list:** An ordered, mutable collection of items.

```
colors = ["red", "green", "blue"]
```

- **tuple**: An ordered, immutable collection of items.

```
coordinates = (10, 20, 30)
```

- **range**: immutable sequence of numbers, commonly used for looping a specific number of times in for loops.

```
range(start, stop, step) # syntax

# Example
for i in range(1, 10, 2):
    print(i)
```

  - start (optional) – The starting value (default is 0).
  - stop – The end value (not included in the range).
  - step (optional) – The increment (default is 1).

3. Set Types

- **set**: An unordered collection of unique items.

```
unique_numbers = {1, 2, 3, 3}  # Output: {1, 2, 3}
```

- **frozenset**: An immutable version of a set.

```
immutable_set = frozenset({1, 2, 3})
```

4. Mapping Types

- **dict (Dictionary)**: A collection of key-value pairs.

```
person = {"name": "Alice", "age": 25}
```

5. Boolean Type

- **bool**: Represents True or False values.

```
is_active = True
```

6. None Type

- **NoneType**: Represents a null or no value.

```
result = None
```

➢ Type Checking and Conversion

- Use the type() function to check the type of a variable.

```
x = 42
print(type(x))  # Output: <class 'int'>
```

- Convert between types using functions like `int()`, `float()`, `str()`, etc.

```
num = "123"
num_int = int(num)  # Converts string to integer
```

➢ Exercise 4: Create Variables of Different Types

Problem Statement: Create variables of different data types (int, float, str, list, tuple, set, dict, bool) and print their values.

Solution:

```
x = 42  # int
y = 3.14  # float
name = "Python"  # str
colors = ["red", "green", "blue"]  # list
coordinates = (10, 20, 30)  # tuple
unique_numbers = {1, 2, 3}  # set
person = {"name": "Alice", "age": 25}  # dict
is_active = True  # bool

# Print values
print("Integer:", x)
print("Float:", y)
print("String:", name)
print("List:", colors)
print("Tuple:", coordinates)
print("Set:", unique_numbers)
print("Dictionary:", person)
print("Boolean:", is_active)
```

➢ Exercise 5: Check Variable Types

Problem Statement: Write a program to check the type of a variable using type().

Solution:

```
x = 42
name = "Python"
is_active = True

print("Type of x:", type(x))
print("Type of name:", type(name))
print("Type of is_active:", type(is_active))
```

➢ Exercise 6: String to Integer and Float Conversion

Problem Statement: Convert a string containing a number (e.g., "45") into an integer and a float.

Solution:

```
num_str = "123"
num_int = int(num_str)  # Convert to integer
num_float = float(num_str)  # Convert to float

print("String:", num_str)
print("Integer:", num_int)
print("Float:", num_float)
```

➢ Exercise 7: Sum of List Elements

Problem Statement: Create a list of numbers and calculate their sum using the sum() function.

Solution:

```
numbers = [10, 20, 30, 40]
result = sum(numbers)

print("Numbers:", numbers)
print("Sum:", result)
```

➢ Exercise 8: Dictionary Operations

Problem Statement: Define a dictionary with at least three key-value pairs and display its keys and values.

Solution:

```
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
```

```
print("Keys:", person.keys())

print("Values:", person.values())
```

❖ Numbers in Python

Python provides robust support for numerical data types and operations. Numbers in Python can be categorized into different types:

1. **Integer (int):**
   - Whole numbers, positive or negative, without decimals.
   - Examples: 5, -10, 0.
2. **Floating Point (float):**
   - Numbers with decimal points or in exponential form.
   - Examples: 3.14, -0.001, 1.2e3.
3. **Complex Numbers (complex):**
   - Numbers with a real and an imaginary part.
   - Represented as a + bj, where a is the real part and b is the imaginary part.
   - Example: 3 + 4j.

➢ Key Numerical Operations

Python supports standard arithmetic operations and built-in functions for numerical computations.

- **Arithmetic Operators:**

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	2 * 3	6
/	Division	8 / 2	4.0
//	Floor Division	7 // 2	3
%	Modulus (Remainder)	10 % 3	1
**	Exponentiation	2 ** 3	8

- **Built-in Functions for Numbers:**

Function	Description	Example	Result
abs(x)	Absolute value of x	abs(-7)	7
pow(x, y)	x raised to the power y	pow(2, 3)	8
round(x)	Rounds x to nearest integer	round(4.6)	5
int(x)	Converts x to an integer	int(4.9)	4
float(x)	Converts x to a float	float(7)	7.0
complex(a, b)	Creates complex number a + bj	complex(3, 4)	3 + 4j

➢ Number Type Conversion

Python allows type casting between different numerical types:

- int(x) → Converts x to an integer.
- float(x) → Converts x to a floating-point number.
- complex(x, y) → Converts x and y to a complex number (y defaults to 0 if not provided).

Examples of Number Usage

```
# Example 1: Basic Arithmetic
x = 10
y = 3
print("Addition:", x + y)       # Output: 13
print("Exponentiation:", x ** y) # Output: 1000

# Example 2: Type Conversion
z = 7.5
print("Integer Conversion:", int(z)) # Output: 7

# Example 3: Complex Numbers
c = complex(2, 3)
print("Real Part:", c.real)          # Output: 2.0
print("Imaginary Part:", c.imag)    # Output: 3.0
```

Key Points to Remember

- Division / always returns a float, even if both operands are integers.
- Floor division // truncates the decimal part and returns an integer.
- Python's numerical operations handle large integers gracefully without overflow errors.
- Complex numbers are not supported in all mathematical libraries; use **cmath** for advanced operations.

➢ Exercise 9: Arithmetic Operations

Problem Statement: Write a Python program that performs the following operations on two user-provided numbers:

1. Calculates and prints their sum, difference, product, and division result.
2. Finds and displays the remainder when the first number is divided by the second.
3. Calculates and prints the result of raising the first number to the power of the second.

Solution:

```
# Input two numbers from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Perform arithmetic operations
print("Sum:", num1 + num2)
print("Difference:", num1 - num2)
print("Product:", num1 * num2)
print("Division:", num1 / num2)
print("Remainder:", num1 % num2)
print("Exponentiation:", num1 ** num2)
```

➢ Exercise 10: Type Conversion and Complex Numbers

Problem Statement: Write a Python program to:

1. Convert a floating-point number to an integer and vice versa.
2. Create a complex number from two user inputs (real and imaginary parts) and display its real and imaginary components.
3. Calculate the conjugate of the complex number.

Solution:

```
# Input a floating-point number
float_num = float(input("Enter a floating-point number: "))

# Convert to integer
int_num = int(float_num)
print("Integer value:", int_num)

# Convert back to float
converted_float = float(int_num)
```

```
print("Converted back to float:", converted_float)

# Input real and imaginary parts for a complex number
real_part = float(input("Enter the real part of the complex
number: "))
imaginary_part = float(input("Enter the imaginary part of the
complex number: "))

# Create and display the complex number
complex_num = complex(real_part, imaginary_part)
print("Complex number:", complex_num)
print("Real part:", complex_num.real)
print("Imaginary part:", complex_num.imag)

# Calculate and display the conjugate
print("Conjugate of the complex number:", complex_num.conjugate())
```

<u>Practical 1:</u> Collecting and Displaying Student Details

Problem Statement: Write a Python program that **Problem Statement:** Write a Python program that collects basic student details and displays them in a structured format. The program should:

1. Ask the user to enter their **full name, age, course preference, and expected graduation year**.
2. Store the data in **variables**.
3. Use **arithmetic operations** to calculate the number of years left until graduation.
4. Print the details in a **well-formatted output**, displaying:

```
------ Student Enrollment Form ------
Name: Patel Harsh
Age: 18
Course: Computer Engineering
Expected Graduation Year: 2028
Years Left Until Graduation: 4
-------------------------------------
```

Objective:

- To understand how to take user input in Python.
- To store and manipulate variables.
- To perform basic arithmetic operations.
- To display formatted output using print statements.

Theory (Recap):

Python provides various ways to collect and manipulate user input. The input() function is used to receive input from the user, which is stored as a string. Numeric data can be converted using int() or float(). Basic arithmetic operations such as subtraction (-) can be used to calculate values like the years left until graduation. Python also supports formatted output using multiple print() statements.

Key concepts used in this program:

1. **Input Handling:** input() function.
2. **Data Types:** Strings and Integers.
3. **Arithmetic Operations:** Subtraction.
4. **Formatted Output:** Using multiple print() statements.

Procedure:

1. Start the Python script.
2. Use the input() function to collect the student's full name, age, course preference, and expected graduation year.
3. Convert the numeric input values (age and graduation year) to integers using int().
4. Compute the years left until graduation by subtracting the current age from the expected graduation year.
5. Display the collected details in a structured format using multiple print() statements.
6. Execute the program and verify the output.

Implementation and Result:

1. Students are required to implement this practical on their own using the above procedure.
2. After successful execution, students must:
    - Manually write the source code (Python) in their record book.
    - Write the expected output in their record book.
3. Save the Python files and results in the college server at the following location:
    - **Server Path:** \\10.10.30.20
    - **Folder:** New Anonymous Share > 103SB CW-2 > Division Folder (e.g., D12 or D03) > Batch Folder
    - Create a folder inside the batch folder with your **Enrollment ID** (e.g., 24CP01).
    - Inside your folder, create a subfolder named **Python > Practical_1**.
    - Save the Python file in this folder.

Conclusion:

Students must write the conclusion in their record book themselves based on their experience and understanding of the practical.

❖ Strings in Python

Strings in Python are sequences of characters enclosed in single quotes ('), double quotes ("), or triple quotes (''' or """). They are immutable, meaning once a string is created, it cannot be modified.

➢ Creating Strings

Strings can be defined in the following ways:

```
# Single-quoted string
a = 'Hello'

# Double-quoted string
b = "World"

# Triple-quoted string for multi-line text
c = '''This is
a multi-line string.'''

print(a, b, c)
```

➢ Accessing Characters in Strings

Characters in a string can be accessed using indexing and slicing:

- **Indexing: s[index]** retrieves the character at the specified position.
  - Example: **s[0]** gives the first character.
- **Slicing: s[start:end:step]** retrieves a portion of the string.
  - Example: **s[1:5]** retrieves characters from index 1 to 4.

```
s = "Python"
print(s[0])     # Output: P
print(s[1:4])   # Output: yth
print(s[-1])    # Output: n (last character)
```

➢ String Operations

Operation	Syntax	Example	Result
Concatenation	str1 + str2	'Hello' + 'World'	'HelloWorld'
Repetition	str * n	'Hi' * 3	'HiHiHi'
Length	len(str)	len('Python')	6
Membership	substr in str	'Py' in 'Python'	True
String Comparison	str1 == str2	'abc' == 'ABC'	False

➢ String Methods

Method	Description	Example	Result
lower()	Converts string to lowercase	'Python'.lower()	'python'
upper()	Converts string to uppercase	'python'.upper()	'PYTHON'
strip()	Removes leading and trailing spaces	' hello '.strip()	'hello'
replace(old, new)	Replaces occurrences of a substring	'hello world'.replace('world', 'Python')	'hello Python'
split(delim)	Splits string into a list	'a,b,c'.split(',')	['a', 'b', 'c']
join(iterable)	Joins elements of a list into a string	','.join(['a', 'b', 'c'])	'a,b,c'
find(sub)	Finds first occurrence of a substring	'hello'.find('e')	1
startswith()	Checks if string starts with a substring	'Python'.startswith('Py')	True
endswith()	Checks if string ends with a substring	'Python'.endswith('on')	True

➢ String Formatting

1. Using format() Method:

```
name = "Alice"
age = 25
print("My name is {} and I am {} years old.".format(name, age))
```

2. **Using f-Strings** (Python 3.6+):

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

➢ Escape Characters

Escape Sequence	Description
\n	Newline
\t	Tab
\\	Backslash
\'	Single Quote
\"	Double Quote

Examples of String Usage :-

```
# Example 1: Basic String Operations
s = "Hello, Python!"
print(s.upper())            # Output: HELLO, PYTHON!
print(s.lower())            # Output: hello, python!
print(s.replace("Python", "World"))  # Output: Hello, World!

# Example 2: String Slicing
print(s[7:13])              # Output: Python

# Example 3: f-String Formatting
name = "Bob"
age = 30
print(f"Name: {name}, Age: {age}")

# Example 4: Escape Characters
print("He said, \"Python is fun!\"")  # Output: He said, "Python
is fun!"
```

➢ Key Points to Remember

- Strings are immutable.
- Indexing starts at 0 and negative indices can be used to access elements from the end.
- String methods do not modify the original string but return a new string.
- Use triple quotes for multi-line strings or strings containing both single and double quotes.

❖ Date and Time in Python

Handling date and time is essential in programming for applications like logging, scheduling, or measuring durations. Python provides the datetime module to manage date and time efficiently.

➢ Importing the datetime Module

To work with date and time, you need to import the **datetime** module:

```
import datetime
```

➢ Getting the Current Date and Time

The datetime module provides methods to retrieve the current date and time:

- **datetime.datetime.now()** gives the current date and time.
- **datetime.date.today()** gives the current date without the time.

```
import datetime

# Get current date and time
dt_now = datetime.datetime.now()
print("Current Date and Time:", dt_now)

# Get only the current date
current_date = datetime.date.today()
print("Current Date:", current_date)
```

➢ Creating Date and Time Objects

You can create custom date or time objects using the datetime class:

- **datetime.date(year, month, day)**
- **datetime.time(hour, minute, second)**
- **datetime.datetime(year, month, day, hour, minute, second)**

```
import datetime

# Create a date object
custom_date = datetime.date(2024, 12, 25)
print("Custom Date:", custom_date)

# Create a time object
custom_time = datetime.time(14, 30, 15)
print("Custom Time:", custom_time)

# Create a datetime object
custom_datetime = datetime.datetime(2024, 12, 25, 14, 30, 15)
print("Custom Date and Time:", custom_datetime)
```

➢ Formatting Date and Time

Use the strftime method to format dates and times into readable strings:

Directive	Meaning	Example Output
%Y	Year (4 digits)	2024
%m	Month (01-12)	12
%d	Day of the month	25
%H	Hour (00-23)	14
%M	Minute (00-59)	30
%S	Second (00-59)	15
%A	Weekday name	Monday
%B	Month name	December

```
import datetime

now = datetime.datetime.now()
formatted = now.strftime("%A, %B %d, %Y %H:%M:%S")
print("Formatted Date and Time:", formatted)
```

➢ Parsing Strings into Date/Time Objects

Use the strptime method to parse strings into date or datetime objects:

```
import datetime
```

```
# Parse a date string
date_string = "2024-12-25"
date_object = datetime.datetime.strptime(date_string, "%Y-%m-%d")
print("Parsed Date Object:", date_object)
```

➢ Date Arithmetic

The **timedelta** class allows you to perform date and time arithmetic, such as adding or subtracting days:

```
import datetime

# Current date
today = datetime.date.today()

# Add 7 days
a_week_later = today + datetime.timedelta(days=7)
print("A Week Later:", a_week_later)

# Subtract 30 days
thirty_days_ago = today - datetime.timedelta(days=30)
print("Thirty Days Ago:", thirty_days_ago)
```

➢ Getting Components of a Date/Time

You can extract individual components (year, month, day, etc.) from a date or datetime object:

```
import datetime

now = datetime.datetime.now()
print("Year:", now.year)
print("Month:", now.month)
print("Day:", now.day)
print("Hour:", now.hour)
print("Minute:", now.minute)
print("Second:", now.second)
```

➢ Key Points to Remember

- datetime module is used for working with dates and times.
- **strftime** is used to format dates into readable strings.
- **strptime** parses strings into date objects.
- **timedelta** enables arithmetic on dates and times.

➢ <u>Exercise 11:</u> Formatting and Calculations with Date and Time

Problem Statement: Write a Python program to perform the following:

1. Display the current date and time in the format: Weekday, Month Day, Year Hour:Minute:Second.
2. Calculate the number of days left until the user-specified date (e.g., New Year).
3. Add 45 days to the current date and display the result in YYYY-MM-DD format.

Solution:

```
import datetime

# 1. Display current date and time in the specified format
now = datetime.datetime.now()
formatted_now = now.strftime("%A, %B %d, %Y %H:%M:%S")
print("Current Date and Time:", formatted_now)

# 2. Calculate days until user-specified date
user_date = input("Enter a target date (YYYY-MM-DD): ")
target_date = datetime.datetime.strptime(user_date, "%Y-%m-%d")
days_left = (target_date - now).days
print("Days left until target date:", days_left)

# 3. Add 45 days to the current date
future_date = now + datetime.timedelta(days=45)
print("Date after 45 days:", future_date.strftime("%Y-%m-%d"))
```

❖ Basic Operators in Python

Operators in Python are special symbols or keywords that perform operations on operands. They are classified into several types based on the operation they perform.

➢ Types of Operators

1. Arithmetic Operators

Used to perform basic mathematical operations.

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

Example:-

```
x = 10
y = 3

print(x + y)   # Output: 13
print(x - y)   # Output: 7
print(x * y)   # Output: 30
print(x / y)   # Output: 3.333...
print(x % y)   # Output: 1
print(x ** y)  # Output: 1000
print(x // y)  # Output: 3
```

2. Comparison (Relational) Operators

Used to compare two values. The result is a Boolean value (True or False).

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

Example:

```
a = 10
b = 20

print(a == b)  # Output: False
print(a != b)  # Output: True
print(a > b)   # Output: False
print(a < b)   # Output: True
print(a >= b)  # Output: False
print(a <= b)  # Output: True
```

3. Logical Operators

Used to perform logical operations. The result is a Boolean value.

Operator	Description	Example	Result
and	Returns True if both are True	True and False	False
or	Returns True if at least one is True	True or False	True
not	Reverses the Boolean value	not True	False

Example:

```
x = True
y = False

print(x and y)  # Output: False
print(x or y)   # Output: True
print(not x)    # Output: False
```

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example	Result
=	Assign	x = 5	x = 5
+=	Add and assign	x += 3	x = x + 3
-=	Subtract and assign	x -= 3	x = x - 3
*=	Multiply and assign	x *= 3	x = x * 3
/=	Divide and assign	x /= 3	x = x / 3
%=	Modulus and assign	x %= 3	x = x % 3
//=	Floor divide and assign	x //= 3	x = x // 3
**=	Exponentiation and assign	x **= 3	x = x ** 3
&=	Bitwise AND and assign	x &= 3	x = x & 3
|=	Bitwise OR and assign	x |= 3	x = x | 3
^=	Bitwise XOR and assign	x ^= 3	x = x ^ 3
>>=	Right shift and assign	x >>= 3	x = x >> 3
<<=	Left shift and assign	x <<= 3	x = x << 3
:=	Walrus operator (assign and return)	print(x := 3)	Prints 3 and assigns 3 to x

Example:

```
x = 5
x += 3  # Equivalent to x = x + 3
print(x)  # Output: 8
x -= 3 # Equivalent to x = x - 3 ,here the x = 8
print(x)  # Output: 5
x *= 3 # Equivalent to x = x * 3 ,here the x = 5
print(x)  # Output: 15
x /= 3 # Equivalent to x = x * 3 ,here the x = 15
print(x)  # Output: 5.0
```

5. Bitwise Operators

Used to perform operations on binary numbers.

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

Example:

```
a = 5  # Binary: 101
b = 3  # Binary: 011

print(a & b)  # Output: 1 (Binary: 001)
print(a | b)  # Output: 7 (Binary: 111)
print(a ^ b)  # Output: 6 (Binary: 110)
print(~a)     # Output: -6 (Binary: ...11010)
print(a << 2) # Output: 20 (Binary: 10100)
print(a >> 2) # Output: 1 (Binary: 001)
```

6. Membership Operators

Used to test membership in sequences like strings, lists, or tuples.

Operator	Description	Example	Result
in	Returns True if found	'a' in 'apple'	True
not in	Returns True if not found	'x' not in 'apple'	True

Example:

```
x = 'apple'
print('a' in x)      # Output: True
print('x' not in x)  # Output: True
```

7. Identity Operators

Used to compare memory locations of two objects.

Operator	Description	Example	Result
is	Returns True if same object	x is y	True/False
is not	Returns True if not same object	x is not y	True/False

Example:

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)       # Output: True (same object)
print(x is z)       # Output: False (different objects)
print(x == z)       # Output: True (same values)
```

➢ Exercise 12: Arithmetic and Logical Operators

Problem Statement : Write a Python program that takes three numbers as input from the user. Perform the following operations:

1. Compute the sum, product, and difference of the numbers.
2. Check if all the numbers are positive using a logical operator.
3. Determine if the sum of the numbers is greater than 100.

Solution

```
# Input three numbers from the user
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
num3 = int(input("Enter the third number: "))

# Perform arithmetic operations
sum_of_numbers = num1 + num2 + num3
product_of_numbers = num1 * num2 * num3
difference = num1 - num2 - num3

# Display results
print("Sum of numbers:", sum_of_numbers)
print("Product of numbers:", product_of_numbers)
print("Difference of numbers:", difference)

# Logical operations
all_positive = (num1 > 0) and (num2 > 0) and (num3 > 0)
print("All numbers are positive:", all_positive)

# Check if sum is greater than 100
is_sum_greater = sum_of_numbers > 100
print("Sum greater than 100:", is_sum_greater)
```

➢ Exercise 13: Bitwise and Identity Operators

Problem Statement: Develop a Python program that:

1. Performs bitwise AND, OR, and XOR operations on two user-provided integers.
2. Verifies if two variables point to the same object in memory.

Solution

```
# Input two integers from the user
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))

# Bitwise operations
print("Bitwise AND:", num1 & num2)
print("Bitwise OR:", num1 | num2)
print("Bitwise XOR:", num1 ^ num2)
print("Bitwise NOT (num1):", ~num1)
print("Bitwise Left Shift (num1 by 2):", num1 << 2)
print("Bitwise Right Shift (num2 by 2):", num2 >> 2)

# Identity operation
x = [num1]
y = [num1]
z = x

print("x is y:", x is y)  # Should be False
print("x is z:", x is z)  # Should be True
print("x == y:", x == y) # Should be True (same value)
```

Practical: 2 Automated Billing System for a Coffee Shop

Problem Statement: Design a Python program that simulates an automated billing system for a coffee shop. The program should:

1. Display a menu with prices (e.g., Coffee - ₹70, Tea - ₹50, Sandwich - ₹100).
2. Ask the user to enter the item name and quantity.
3. Calculate the total bill using arithmetic operators. Apply 18% GST on the total. Display final bill amount after tax. (For computation, do not use if statements or any other concepts beyond what has been covered so far.)
4. Display the current date and time when generating the bill.
5. Format and display the bill in a structured manner using string manipulation.

Objective:

- To understand how to take user input in Python.
- To perform arithmetic operations.
- To display formatted output using string manipulation.
- To use Python's built-in functions for date and time handling.

Theory:

Python provides various functionalities to create a simple billing system using basic concepts such as:

- User Input: input() function to collect item names and quantities.
- Data Storage: Variables to store item prices and calculations.
- Arithmetic Operations: Multiplication for total calculation and percentage computation for tax.
- String Formatting: Displaying the bill in a structured format.
- Date and Time: The datetime module is used to fetch the current date and time.

Key concepts used in this program:

1. Input Handling: input() function.
2. Data Types: Strings, Integers, and Floats.
3. Arithmetic Operations: Multiplication and Addition.
4. String Manipulation: Formatting output.
5. Date and Time Handling: datetime.now() from the datetime module.

Procedure:

1. Start the Python script.
2. Display the menu with item prices.
3. Use the input() function to get the item name and quantity.
4. Define prices for menu items and calculate the total amount using arithmetic operators.
5. Compute the final bill amount by adding 18% GST.
6. Retrieve the current date and time.
7. Format and print the structured bill.

Implementation and Result:

1. Students are required to implement this practical on their own using the above procedure.
2. After successful execution, students must:
    - Manually write the source code (Python) in their record book.
    - Write the expected output in their record book.
3. Save the Python files and results in the college server at the following location:
    - **Server Path:** \\10.10.30.20
    - **Folder:** New Anonymous Share > 103SB CW-2 > Division Folder (e.g., D12 or D03) > Batch Folder
    - Create a folder inside the batch folder with your **Enrollment ID** (e.g., 24CP01).
    - Inside your folder, create a subfolder named **Python > Practical_2**.
    - Save the Python file in this folder.

Conclusion:

Students must write the conclusion in their record book themselves based on their experience and understanding of the practical.

❖ Control Structures

Control structures in Python dictate the flow of execution within a program. They include:

1. Sequential Control
2. Selection Control (Decision Making)
3. Iteration Control (Loops)

1. Sequential Control

Sequential control means that statements execute one after another in the order they appear.

Example:

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

Here, each statement is executed sequentially.

2. Selection Control (Decision Making)

Selection control allows decision-making by executing different blocks of code based on conditions.

a) if Statement

The if statement in Python is a fundamental control structure used for decision-making. It allows a program to execute a block of code only when a specified condition evaluates to True. If the condition is False, the block is skipped.An if statement is written by using the **if** keyword.

Syntax:

```
if condition:
    # Indented block executes if condition is True
    statement(s)
```

Python uses indentation (typically 4 spaces) to define the block of code that belongs to the if statement.

How Conditions Work: A condition in an if statement is an expression that evaluates to a Boolean value (True or False). Python considers the following values as False (Falsy values):

- None
- False
- 0 (Integer or Float)
- "" (Empty String)
- [] (Empty List)
- {} (Empty Dictionary)
- () (Empty Tuple)
- set() (Empty Set) All other values are considered True (Truthy values).

Example:

```
if 50:
    print("This statement will execute.")

# Output:
This statement will execute.
```

Here, 50 is a non-zero number, so it is considered True.

- Single Condition if Statement:

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

Since x > 5 evaluates to True, the print statement executes.

- if with a False Condition:

```
x = 0
if x:
    print("This won't be printed")
```

Since x = 0, which is a Falsy value, the print statement is not executed.

- Using Comparison Operators in if:

```
x = 10
y = 20
if x < y:
    print("x is less than y")
```

Python supports various comparison operators in if conditions, such as:

- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

- Logical Operators in if: Logical operators (and, or, not) can be used to combine multiple conditions.

```
x = 15
y = 10
if x > 10 and y < 20:
    print("Both conditions are True")
```

- if with Membership Operators: Membership operators (in, not in) check if a value exists in a sequence.

```
name = "Python"
if 'y' in name:
    print("'y' is present in the string")
```

- Nested if Statements: An if statement inside another if statement is called a nested if.

```
num = 10
if num > 0:
    if num % 2 == 0:
        print("Positive even number")
```

b) if-else Statement

The if-else statement in Python is used when a program needs to execute one block of code if a condition is True and a different block if the condition is False. It extends the basic if statement by providing an alternative path of execution.

Syntax

```
if condition:
    # Executes if the condition is True
    statement(s)
else:
    # Executes if the condition is False
    statement(s)
```

The else block must be indented at the same level as the corresponding if block.

Example:

```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print("Even Number")
else:
    print("Odd Number")
```

- If the input number is divisible by 2, it is even, and the if block executes.
- Otherwise, the else block runs, printing that the number is odd.

- Nested if-else: An if-else statement inside another if-else is called a nested if-else.

```
num = int(input("Enter a number: "))
if num > 0:
    if num % 2 == 0:
        print("Positive Even Number")
    else:
        print("Positive Odd Number")
else:
    print("Negative Number")
```

- If num > 0, it enters the nested if.
- The nested if further checks if the number is even or odd.
- If num is not greater than zero, the else block executes.

- Short Hand If ... Else: If you have only one statement to execute, one for if, and one for else, you can put it all on the same line

```
a = 11
b = 510

print("A") if a > b else print("B")
```

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

```
a = 221
b = 221

print("A") if a > b else print("=") if a == b else
print("B")
```

c) if-elif-else Statement

The if-elif-else statement in Python is used for multiple conditional checks. It allows a program to evaluate multiple conditions sequentially and execute the block of code corresponding to the first True condition. If none of the conditions are met, the else block executes.

Syntax

```
if condition1:
    # Executes if condition1 is True
    statement(s)
elif condition2:
    # Executes if condition2 is True (only if condition1 is
False)
    statement(s)
else:
    # Executes if none of the above conditions are True
    statement(s)
```

Example:

```
num = int(input("Enter a number: "))
if num > 0:
    print("Positive Number")
elif num < 0:
    print("Negative Number")
else:
    print("Zero")
```

- If num is greater than 0, the first if block executes.
- If num is less than 0, the elif block executes.
- If neither condition is met, the else block executes.

Example of Multiple elif Conditions:

```
marks = int(input("Enter your marks: "))
if marks >= 90:
    print("Grade: A")
elif marks >= 80:
    print("Grade: B")
elif marks >= 70:
    print("Grade: C")
elif marks >= 60:
```

```
    print("Grade: D")
else:
    print("Grade: F")
```

- The program evaluates the marks and assigns a grade accordingly.
- It stops checking further conditions once a True condition is met.

➢ Exercise 14: if Statement

Problem: Write a Python program that asks the user for their age. If the age is 18 or above, print "**You are eligible to vote**".

Solution:

```
age = int(input("Enter your age: "))
if age >= 18:
    print("You are eligible to vote")
```

➢ Exercise 15: if-else Statement

Problem: Write a Python program that asks the user for their exam score. If the score is 40 or more, print "Pass", otherwise print "Fail".

Solution:

```
score = int(input("Enter your exam score: "))
if score >= 40:
    print("Pass")
else:
    print("Fail")
```

➢ Exercise 16: if-elif Statement

Problem: Write a Python program that asks the user for the temperature in Celsius and classifies it as "Cold", "Warm", or "Hot" based on the following conditions:

- Below 15°C: Cold
- 15°C to 30°C: Warm
- Above 30°C: Hot

Solution:

```
temp = float(input("Enter temperature in Celsius: "))
if temp < 15:
    print("Cold")
elif temp <= 30:
    print("Warm")
else:
    print("Hot")
```

➢ Exercise 17: Nested if-else Statement

Problem: Write a Python program that asks the user for a year and checks whether it is a leap year or not. A year is a leap year if:

- It is divisible by 4, and
- If it is a century year (ending in 00), it must also be divisible by 400.

Solution:

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

if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print(year, "is a leap year")
        else:
            print(year, "is not a leap year")
    else:
        print(year, "is a leap year")
else:
    print(year, "is not a leap year")
```

Practical:3 University Admission Eligibility Check

Problem Statement:

Write a Python program that checks if a student is eligible for university admission. The program should:

1. Ask the user to enter their percentage marks in three subjects.
2. Calculate the average marks.
3. If the average is above 75%, print "**Eligible for Admission**"; otherwise, print "**Not Eligible**".
4. If the student scores above 90%, display a message saying "**You qualify for a scholarship!**".
5. Handle edge cases (e.g., if the user enters marks below 0 or above 100, display an error message).

Objective:

- To understand conditional statements (if, if-else, if-elif).
- To implement user input validation in Python.
- To apply mathematical calculations (average calculation).
- To develop a program that handles real-world admission criteria.

Theory (Recap):

1. Conditional Statements (if, if-else, if-elif)
    - Used to make decisions based on conditions.
    - if executes a block of code if the condition is true.
    - if-else provides an alternative action if the condition is false.
    - if-elif allows checking multiple conditions sequentially.
2. User Input and Validation
    - input() is used to take user input, which needs to be converted to float for calculations.
    - Input validation ensures marks are between 0 and 100 before processing.
3. Mathematical Calculation (Average Marks)
    - The average is calculated using:

$$\text{average} = \frac{\text{subject1} + \text{subject2} + \text{subject3}}{3}$$

- If the average is above 75%, the student is eligible for admission.
- If the average is above 90%, they qualify for a scholarship.

Procedure:

1. Ask the user to enter marks for three subjects.
2. Validate the input to ensure marks are between 0 and 100.
3. Compute the average of the three marks.
4. Use conditional statements:
    - If the average is above 75%, print "Eligible for Admission".
    - Otherwise, print "Not Eligible".
    - If the average is above 90%, print "You qualify for a scholarship!".
5. Handle invalid input cases and display an appropriate error message.

Implementation and Result:

1. Students are required to implement this practical on their own using the above procedure.
2. After successful execution, students must:
    - Manually write the source code (Python) in their record book.
    - Write the expected output in their record book.
3. Save the Python files and results in the college server at the following location:
    - **Server Path:** \\10.10.30.20
    - **Folder:** New Anonymous Share > 103SB CW-2 > Division Folder (e.g., D12 or D03) > Batch Folder
    - Create a folder inside the batch folder with your **Enrollment ID** (e.g., 24CP01).
    - Inside your folder, create a subfolder named **Python > Practical_3**.
    - Save the Python file in this folder.

Conclusion:

Students must write the conclusion in their record book themselves based on their experience and understanding of the practical.

3. Iteration Control (Loops)

Iteration control, also known as loops, allows a program to execute a block of code multiple times. Python provides two main types of loops:

1. **for loop** – Used for iterating over a sequence (e.g., list, tuple, dictionary, string, or range).
2. **while loop** – Repeats execution as long as a condition remains True.

Loops help automate repetitive tasks and essential in programming, as they help reduce redundancy, improve efficiency, and enhance code readability.

1. The for Loop

The for loop in Python is used for iterating over sequences (like lists, tuples, dictionaries, sets, or strings). It executes the block of code once for each element in the sequence.

Syntax:

```
for variable in sequence:
    # loop body
```

- The loop iterates over each element in the sequence.
- The variable takes on the value of each element in the sequence, one at a time.
- The block of code inside the loop executes for each element in the sequence.

Example 1: Iterating Over a List

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

Example 2: Using range() with for Loop

The range() function is commonly used to generate a sequence of numbers in loops.

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

#Output
0
1
2
3
4
(The range(5) generates numbers from 0 to 4 (excluding 5).)
```

Example 3: Using for Loop with a String

```
word = "Python"
for letter in word:
    print(letter)

#Output
P
y
t
h
o
n
```

Example 4: Iterating Over a Dictionary

```
student = {"name": "John", "age": 20, "course": "Computer
Science"}
for key, value in student.items():
    print(key, ":", value)

#Output
name : John
age : 20
course : Computer Science
```

2. The while Loop

The while loop executes a block of code as long as the given condition is True.

Syntax:

```
while condition:
    # loop body
```

- The condition is evaluated before each iteration.
- If the condition is True, the block of code executes.
- The loop continues until the condition becomes False.

Example 1: Basic while Loop

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

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

Example 2: Using while Loop for User Input

```
number = int(input("Enter a number greater than 0: "))
while number <= 0:
    print("Invalid input! Try again.")
    number = int(input("Enter a number greater than 0: "))
print("You entered:", number)

# The loop keeps running until the user enters a number
greater than 0.
```

Example 3: Infinite Loop (Avoiding Pitfalls)

```
while True:
    print("This is an infinite loop!")
```

- Warning: This loop runs indefinitely because the condition True never changes.
- Solution: Use **break** to exit the loop when a specific condition is met

➢ Loop Control Statements

Python provides special control statements to modify loop behavior.

1. break Statement

The break statement is used to exit the loop prematurely when a certain condition is met.

Example: Using break in a while Loop

```
num = 1
while num <= 10:
    if num == 5:
        break
    print(num)
    num += 1

#Output
1
2
3
4
(The loop terminates when num reaches 5.)
```

2. continue Statement

The continue statement skips the current iteration and moves to the next iteration of the loop.

Example: Using continue in a for Loop

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

#Output
1
2
4
5
(When num is 3, the continue statement skips that iteration.)
```

3. else Clause in Loops

Python allows an else block to be used with loops. The else block executes after the loop completes normally (without a break).

Example: Using else with a for Loop

```
for i in range(5):
    print(i)
else:
    print("Loop completed successfully!")

#Output
0
1
2
3
4
Loop completed successfully!
```

Example: Using else with a while Loop

```
num = 1
while num < 5:
    print(num)
    num += 1
else:
    print("Loop finished normally!")

#Output
1
2
3
4
Loop finished normally!
```

➢ Nested Loops

A loop inside another loop is called a **nested loop**.

Example: Nested for Loop

```
for i in range(1, 4):
    for j in range(1, 4):
        print(f"i={i}, j={j}")

#Output
i=1, j=1
i=1, j=2
i=1, j=3
i=2, j=1
i=2, j=2
i=2, j=3
i=3, j=1
i=3, j=2
i=3, j=3
```

Example: Nested while Loop

```
i = 1
while i <= 3:
    j = 1
    while j <= 3:
        print(f"i={i}, j={j}")
        j += 1
    i += 1
#Output
i=1, j=1
i=1, j=2
i=1, j=3
i=2, j=1
i=2, j=2
i=2, j=3
i=3, j=1
i=3, j=2
i=3, j=3
```

- This structure is useful for working with multi-dimensional data like matrices.

➢ Exercise 18: while Loop

Problem:
Write a Python program that asks the user for a number and calculates the factorial of that number using a while loop.

Solution:

```
num = int(input("Enter a number: "))
factorial = 1
i = 1

while i <= num:
    factorial *= i
    i += 1

print("Factorial of", num, "is", factorial)
```

Practical 4: Basic ATM Simulation

Problem Statement:

Create a Python program that acts as a basic ATM. The program should:

1. Display an initial balance (e.g., ₹5000).
2. Allow the user to withdraw money, ensuring they do not withdraw more than the available balance.
3. Use a loop to allow multiple transactions until the user exits.
4. Deduct the amount and display the remaining balance after each transaction.
5. If the balance goes below ₹100, warn the user about a low balance.

Objective:

- To practice loops for repeated user interactions.
- To implement if-else conditions for checking withdrawal limits.
- To work with user input and numeric calculations.
- To simulate a simple banking system.

Theory (Recap):

- **Loops (while loop):** Used to repeatedly execute a block of code until a specific condition is met. It helps in scenarios where continuous execution is required until the user decides to stop.
- **Conditional Statements (if, if-else, if-elif):** Allow decision-making in programs. These statements check conditions and execute different blocks of code based on whether the condition is True or False.
- **Nested if-else:** When an if or else statement contains another if-else, it helps in handling multiple levels of conditions, ensuring detailed decision-making.
- **User Input Handling:** The input() function is used to accept user data, which can be validated to ensure correct values are processed.
- **Mathematical Calculations:** Variables are updated dynamically based on arithmetic operations, ensuring that computed values are accurate as per given conditions.

Procedure:

1. Initialize the **account balance** (e.g., ₹5000).
2. Use a while loop to allow continuous transactions.
3. Prompt the user to enter a withdrawal amount.
4. Check if the withdrawal is valid:
    - If the withdrawal amount exceeds the balance, display an error.
    - Otherwise, deduct the amount and update the balance.
5. Warn the user if the balance falls below ₹100.
6. Ask the user if they want another transaction or want to exit.

Implementation and Result:

1. Students are required to implement this practical on their own using the above procedure.
2. After successful execution, students must:
    - Manually write the source code (Python) in their record book.
    - Write the expected output in their record book.
3. Save the Python files and results in the college server at the following location:
    - **Server Path:** \\10.10.30.20
    - **Folder:** New Anonymous Share > 103SB CW-2 > Division Folder (e.g., D12 or D03) > Batch Folder
    - Create a folder inside the batch folder with your **Enrollment ID** (e.g., 24CP01).
    - Inside your folder, create a subfolder named **Python > Practical_4**.
    - Save the Python file in this folder.

Conclusion:

Students must write the conclusion in their record book themselves based on their experience and understanding of the practical.