Unit 1

Section No.	Topic	Subtopics Covered
1.1	Introduction to Python	- What is Python?- History of Python- Applications of Python- Key Features of Python
1.2	Installing Python & Using Interpreters	- Installing Python (Windows, macOS, Linux)- Working with IDLE- Using Python in Terminal/Command Prompt- Writing Python in VS Code- Using Jupyter Notebook- PyCharm IDE setup
1.3	Python Syntax & Indentation Rules	- Basic Python syntax- Importance of indentation- Block structure- Comparison with other languages
1.4	Variables & Data Types	- Variable declaration & naming rules- Standard data types: int, float, str, bool, NoneType- Complex numbers- Collections overview (list, tuple, dict, set)- Dynamic typing
1.5	Dynamic Typing & Type Checking	- Python's dynamic nature- type() function- isinstance() function- Examples with different types
1.6	Basic Input/Output	- print() function (formatting, sep, end)- input() function- Reading user input- Combining input with variables
1.7	Type Conversion	- Implicit type conversion- Explicit type conversion- Conversion functions (int(), float(), str(), etc.)- Real-life example: billing system
1.8	Operators in Python	- Arithmetic operators- Relational/comparison operators- Logical operators- Assignment operators- Bitwise operators- Membership operators- Identity operators- Real-life example: calculator
1.9	Control Flow	- if statement- if-else- if-elif-else- Nested if- match-case (Python 3.10+)- for loop- while loop- else with loops
1.10	Loop Control Statements	- break- continue- pass- else with loops- Nested loops with control statements

1.1 Introduction to Python

What is Python?

Python is a **high-level, object-oriented, interpreted, dynamically typed programming language**. Let's break that down:

- **High-level** → You don't need to manage hardware details like memory allocation; Python handles it for you.
- **Object-oriented** → Everything in Python is an object, and it supports concepts like classes, inheritance, and polymorphism.
- **Interpreted** → Python code is executed line by line by the interpreter, making it easier to test and debug.
- **Dynamically typed** → You don't need to declare variable types; Python figures it out during runtime.

👉 In simple terms: Python is like **speaking directly in English to the computer** — no extra translations, and the computer understands your intent quickly.

History of Python

- **1980s (Late)** → Dutch programmer *Guido van Rossum* started developing Python as a hobby project while working at CWI (Centrum Wiskunde & Informatica) in the Netherlands.
- **1991** → Python 0.9.0 was released with important features: functions, exception handling, and core data types (str, list, dict).
- **2000** → Python 2.0 launched with new features but later created compatibility issues.
- **2008** → Python 3.0 introduced with a focus on simplicity and future-proofing (not backward compatible).
- **2020** → Official support for Python 2 ended; Python 3 became the universal standard.

👉 The name *Python* came from the comedy show **"Monty Python's Flying Circus"**, not from the snake 🐍.

Key Features of Python

1. **Readable & Simple Syntax** → Almost like writing English. Example:

   print("Hello, World!")

   This does not require semicolons, braces, or extra setup.

2. **Cross-Platform** → Runs on Windows, Linux, macOS, and even embedded devices.
3. **Rich Standard Library** → Includes modules for file handling, math, networking, date/time, etc.
4. **Huge Ecosystem of Libraries** → NumPy (math), Pandas (data), Django (web), TensorFlow (AI), Flask (APIs).
5. **Extensible & Integrable** → Python can integrate with C, C++, Java, and even run inside other applications.
6. **Interpreted & Interactive** → No compilation step; run code directly. Great for testing small code snippets.
7. **Dynamic Typing** → You don't declare types explicitly:

```
x = 5      # integer
x = "five"  # now a string
```

8. **Object-Oriented but Flexible** → Supports OOP concepts (classes, inheritance) but also works in a procedural or functional style.
9. **Portable & Open Source** → Free to use, modify, and distribute.
10. **Vast Community Support** → Millions of developers contribute solutions, tutorials, and libraries.

Applications of Python

Python powers many modern technologies:

- **Web Development** → Django, Flask.
- **Data Science & ML** → Pandas, NumPy, Scikit-learn, TensorFlow.
- **Automation** → Scripts for file handling, email, testing.
- **AI & NLP** → Chatbots, translators, image recognition.
- **Game Development** → Pygame.
- **IoT** → MicroPython for microcontrollers.
- **Enterprise Applications** → Used by Google, YouTube, Netflix, NASA.

Key Takeaways

- Python = high-level, object-oriented, interpreted, dynamically typed language.
- Simple, readable, and versatile with a vast library ecosystem.
- Popular in **web, AI, ML, automation, data science, IoT, and more**.

1.2 Installing Python & Using Interpreters

1.2.1 Installing Python

Windows

1. Visit https://www.python.org/downloads/.
2. Download the latest Python version (e.g., Python 3.x).
3. Run the installer.
    - Check **“Add Python to PATH”** before proceeding.
    - Choose **Install Now**.
4. After installation, open Command Prompt and type:

```
python --version
or
python3 —version
```

You should see the installed version (e.g., Python 3.12.1).

macOS

- Python 2.x is usually pre-installed. For Python 3:
    1. Install Homebrew.

       Run:

       ```
       brew install python
       ```

    2. Verify:

       ```
       python3 --version
       ```

Linux (Ubuntu/Debian)

1. Open terminal.
2. Run:

```
sudo apt update
sudo apt install python3
```

3. Verify:

```
python3 --version
```

1.2.2 IDLE (Integrated Development and Learning Environment)

- IDLE comes **pre-installed** with Python on Windows and macOS.
- To launch:

  - On **Windows**: Search for **IDLE** in Start menu.
  - On **macOS/Linux**: Run idle3 in terminal (if installed).
- IDLE provides:
  - Python Shell (interactive mode).
  - Simple text editor to write .py files.

👉 *Good for beginners* — lightweight, no setup required.

1.2.3 Running Python in IDLE

1. Open **IDLE**.
2. In the Python Shell, type:

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

3. Press **Enter**.
   Output:

```
Hello, World!
```

👉 For a script file:

- Go to **File → New File**.
- Write:

```
print("Hello from IDLE file!")
```

- Save as hello.py.
- Run → Run Module (F5).

1.2.4 Terminal / Command Prompt

- **Windows**:
  - Open *Command Prompt* → type python or python3.
- **macOS/Linux**:
  - Open *Terminal* → type python3.

This opens the **REPL (Read-Eval-Print Loop)** where you can test code:

```
>>> print("Hello from Python!")
Hello from Python!
```

To run a script saved in a file (e.g., hello.py):

```
python hello.py
```

1.2.4 Installing & Using VS Code

1. Download VS Code from https://code.visualstudio.com/.
2. Install and open VS Code.
3. Install the **Python Extension**:
   - Go to Extensions (Ctrl+Shift+X).
   - Search for “Python” (by Microsoft).
   - Install it.
4. Create a new file hello.py.
5. Select the correct **Python interpreter**:
   - Press Ctrl+Shift+P.
   - Search for **Python: Select Interpreter**.
   - Choose your installed Python version.
6. Run code:
   - Right-click → **Run Python File in Terminal**.

👉 *Best for learners who want a lightweight but powerful editor.*

1.2.5 Installing & Using Jupyter Notebook

1. Install Jupyter via pip:

```
    pip install notebook
```

Start Jupyter:

```
    jupyter notebook
    → Opens in your default browser.
```

4. Create a new Python notebook (.ipynb).
5. Type code in a cell and run with Shift + Enter.

Example:

```
x = [1, 2, 3]
sum(x)
```

Output:

```
6
```

👉 Widely used in **data science and machine learning** because it combines code, results, and notes in one place.

1.2.6 Installing & Using PyCharm

1. Download PyCharm from https://www.jetbrains.com/pycharm/download/.
   - Choose **Community Edition** (free) or **Professional Edition** (paid).
2. Install and launch PyCharm.
3. Create a new project:
   - File → New Project → Select **Python Interpreter** (existing or new virtual environment).
4. Create a Python file (hello.py) inside the project.
5. Write and run your code using the green ▶️ run button.

👉 Great for **professional developers** managing large projects.

Choosing the Right Interpreter

- **Beginners** → IDLE or VS Code.
- **Quick Testing** → Terminal.
- **Data Science/ML** → Jupyter Notebook.
- **Large Applications** → PyCharm.

Key Takeaways

- Python can be installed on Windows, macOS, Linux via official site or package managers.
- Interpreters allow code execution line by line.
- Popular environments: **IDLE, Terminal, VS Code, Jupyter, PyCharm**.
- Choice depends on **learning stage and project size**.

1.3 Python Syntax & Indentation Rules

1.3.1 What is Syntax?

- **Syntax** is the set of rules that defines how programs in a language must be written.
- In Python, syntax is **simple and human-readable** compared to other programming languages.

👉 Example:

```
print("Hello World")   # Valid Python code
```

If we misspell or break syntax rules:

print "Hello World"   # ❌ SyntaxError in Python

1.3.2 Role of Indentation in Python

- Unlike C, Java, or JavaScript where **curly braces { }** are used to define blocks of code, **Python uses indentation (spaces or tabs)**.
- Indentation **defines the scope** of loops, functions, and conditional blocks.

👉 *Analogy*: Indentation in Python is like **paragraph spacing in English essays** — without it, the text is confusing and unreadable.

1.3.3 Indentation Example

Correct code:

```
if True:
    print("Inside if block")
    print("Still inside if block")
print("Outside if block")
```

Output:

```
Inside if block
Still inside if block
Outside if block
```

Incorrect code (missing indentation):

```
if True:
print("This will cause error")
```

Error:

```
IndentationError: expected an indented block
```

1.3.4 Recommended Indentation Style

- **4 spaces per indentation level** (PEP 8 style guide).
- Do not mix tabs and spaces.

Example (Good style):

```
for i in range(3):
    print("Number:", i)
```

Bad style (mixing spaces and tabs) might work in some editors but cause errors elsewhere.

1.3.5 Python Statement Rules

- **Case-sensitive** → print ≠ Print.
- Each statement usually ends with a **newline**, not ;.
    - But you *can* use ; to write multiple statements on one line (not recommended).

Example:

```
x = 10; y = 20; print(x + y)   # Works but not clean
```

Better:

```
x = 10
y = 20
print(x + y)
```

1.3.6 Line Continuation

If a statement is too long, use \ (backslash) or parentheses ().

Example:

```
total = 1 + 2 + 3 + \
        4 + 5 + 6
print(total)
```

Or better (using parentheses):

```
total = (1 + 2 + 3 +
         4 + 5 + 6)
print(total)
```

1.3.7 Comments in Python

- Comments are ignored by the interpreter.
- Used for documentation and readability.
- Single-line → starts with #.
- Multi-line → use triple quotes (''' or """).

Example:

```
# This is a single-line comment
print("Hello")  # Comment after code
```

```
"""
This is a
multi-line comment
"""
```

1.3.8 Example Program (Combining Syntax & Indentation)

```
# Program to check even/odd
num = 7

if num % 2 == 0:
    print(num, "is Even")
else:
    print(num, "is Odd")
```

Output:

```
7 is Odd
```

Key Takeaways

- Python syntax is simple but **strict with indentation**.
- Indentation replaces {} or begin/end keywords.
- Use **4 spaces** consistently, never mix tabs and spaces.
- Comments improve readability but don't affect execution.
- Clean syntax makes Python highly **readable and beginner-friendly**.

Perfect timing love ❤️🐇 let's build **1.4 Variables & Data Types** — in a **learner-friendly, self-study style** with clear definitions, real-world analogies, and code examples. We'll cover **all core Python data types**.

1.4 Variables & Data Types

1.4.1 What are Variables?

- A **variable** is a named storage location in memory that holds data.
- In Python, you don't need to declare type explicitly — just assign a value.

- Variable name rules:
    - Can contain letters, digits, underscore (_).
    - Cannot start with a digit.
    - Case-sensitive (age ≠ Age).
    - Cannot be a reserved keyword (if, class, etc.).

👉 *Analogy*: Think of a variable like a **container** or **label on a box** — the label (variable name) can change, but it always points to some content (value).

1.4.2 Assigning Variables

```
x = 10          # integer
name = "Alice"  # string
pi = 3.14       # float
is_active = True # boolean
```

- Multiple assignment:

```
a, b, c = 1, 2, 3
```

- Same value to multiple variables:

```
x = y = z = 100
```

1.4.3 Python Data Types

Python has **built-in data types** that can be grouped as:

1. Numeric Types

1. **int** → whole numbers

```
age = 25
print(type(age))    # <class 'int'>
```

2. **float** → decimal numbers

```
price = 19.99
```

3. **complex** → complex numbers (useful in math, engineering)

```
z = 3 + 4j
print(z.real, z.imag)   # 3.0 4.0
```

2. Sequence Types

1. **str** → strings (text)

```
message = "Hello World"
```

- Strings are **immutable** (cannot be changed in place).
- Support slicing & indexing:

```
    print(message[0])      # H
    print(message[-1])     # d
    print(message[0:5])    # Hello
```

3. **list** → ordered, mutable collection

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "grape"
print(fruits)
```

4. **tuple** → ordered, immutable collection

```
point = (10, 20)
```

3. Set Types

1. **set** → unordered, unique elements

```
nums = {1, 2, 2, 3, 4}
print(nums)   # {1, 2, 3, 4}
```

2. **frozenset** → immutable set

```
frozen = frozenset([1, 2, 3])
```

4. Mapping Type

- **dict** → key-value pairs

```
student = {"name": "Alice", "age": 21}
print(student["name"])    # Alice
```

5. Boolean Type

- **bool** → True / False

```
is_valid = True
print(5 > 3)    # True
```

6. None Type

- **None** represents “no value” / null.

```
result = None
print(result)    # None
```

7. Binary Types

- **bytes** → immutable sequence of bytes

```
b = b"Hello"
```

- **bytearray** → mutable sequence of bytes

```
ba = bytearray([65, 66, 67])
```

- **memoryview** → memory-efficient view of bytes

```
mv = memoryview(b"Python")
```

1.4.4 Type Checking

- Use type() to check variable type.
- Use isinstance() to check if an object belongs to a class/type.

Example:

```
x = 42
print(type(x))              # <class 'int'>
print(isinstance(x, int))   # True
```

1.4.5 Type Conversion

- **Implicit Conversion (Type Casting by Python)**:
  Python automatically converts smaller data types to larger ones during operations.

```
x = 5       # int
y = 2.5     # float
result = x + y   # 5 + 2.5 → float
print(result)    # 7.5
```

- **Explicit Conversion (Manual Casting)**:

```
a = "100"
b = int(a)   # str → int
print(b + 50)   # 150
```

1.4.6 Real-Life Example

```
# Student information system
student_name = "John"
student_age = 20
is_enrolled = True
marks = [85, 90, 78]

print(f"Name: {student_name}")
print(f"Age: {student_age}")
print(f"Enrolled: {is_enrolled}")
print(f"Average Marks: {sum(marks)/len(marks)}")
```

Output:

```
Name: John
Age: 20
Enrolled: True
Average Marks: 84.33333333333333
```

Key Takeaways

- Variables are **labels for data in memory**.
- Python supports many built-in data types: **Numeric, Sequence, Set, Mapping, Boolean, None, Binary**.
- Use type() and isinstance() to check data types.
- Supports **implicit & explicit type conversion**.
- Strings are immutable, Lists are mutable, Tuples are immutable.
- Choosing the **right data type** improves performance and readability.

1.5 Dynamic Typing & Type Checking

1.5.1 What is Dynamic Typing?

- In some languages (like C, C++ or Java), you must declare a variable's type before using it.

```
int x = 10;   // Java requires type declaration
```

- In Python, **you don't need to declare the type**. The interpreter **decides the type automatically** at runtime based on the value assigned.

```
x = 10         # int
x = "hello"    # now str
```

👉 This is called **Dynamic Typing**.

1.5.2 How Dynamic Typing Works

- Python variables are just **labels (references)** pointing to objects in memory.
- The **type is associated with the object, not the variable**.

Example:

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

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

Analogy: Imagine a **post-it note** (variable name) stuck to a box (value). You can peel it off and stick it on another box anytime.

1.5.3 Advantages of Dynamic Typing

- **Flexibility**: You can reuse variable names for different types.
- **Ease of use**: No need to declare types explicitly.
- **Faster prototyping**: Great for beginners & rapid development.

1.5.4 Disadvantages of Dynamic Typing

- **Errors at runtime** instead of compile-time.
- Harder to catch type-related bugs in large projects.
- May cause **unexpected behavior** if you assume wrong type.

Example:

```
x = 10
x = x + "20"   # ❌ TypeError: unsupported operand type(s)
```

1.5.5 Type Checking in Python

Python provides two main ways to check types:

1. Using type()

- Returns the exact type of a variable.

```
x = 3.14
```

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

2. Using isinstance()

- Checks whether a variable belongs to a specific type/class.
- Supports inheritance checks.

```
x = 42
print(isinstance(x, int))   # True
print(isinstance(x, float)) # False
```

1.5.6 Type Hints (Optional)

- From Python 3.5+, you can add **type hints** for readability and static analysis.
- Python **does not enforce** them at runtime, but tools like *mypy* can check.

Example:

```
def add_numbers(a: int, b: int) -> int:
    return a + b

print(add_numbers(5, 7))       # 12
print(add_numbers("5", "7"))    # 57 (Python doesn’t enforce, but warning if checked)
```

1.5.7 Real-Life Example

```
# Bank account balance example
balance = 1000      # int
print("Balance:", balance)

balance = 1000.50   # float (updated to store decimal)
print("Updated Balance:", balance)

balance = "One Thousand"   # str (unexpected change)
print("Now Balance is:", balance)
```

Output:

```
Balance: 1000
Updated Balance: 1000.5
Now Balance is: One Thousand
```

👉 This shows both the **flexibility** and the **risk** of dynamic typing.

Key Takeaways

- Python is **dynamically typed** → no need to declare type.
- Variables are just **references** to objects; the object decides the type.
- Use type() and isinstance() for type checking.
- **Pros**: Flexible, fast to code.
- **Cons**: Runtime errors, harder to debug in big projects.
- Type hints help improve readability and error detection.

1.6 Basic Input/Output

1.6.1 Input in Python

- Python uses the **input()** function to take user input from the keyboard.
- Input is **always returned as a string**, no matter what the user types.

Syntax:

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

Example:

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

Output:

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

Converting Input into Other Types

Since input() returns a string, we must convert it if we need integers or floats.

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

👉 If you don't convert, operations may cause errors:

```
num = input("Enter a number: ")
print(num + 5)     # ❌ TypeError (string + int)
```

Correct:

```
num = int(input("Enter a number: "))
print(num + 5)
```

1.6.2 Output in Python

- Python uses **print()** to display information.
- By default, it adds a newline after each call.

Examples:

```
print("Hello, World!")
print("Python is fun")
```

Output:

```
Hello, World!
Python is fun
```

Printing Multiple Values

```
name = "Alice"
age = 21
print("Name:", name, "Age:", age)
```

Output:

```
Name: Alice Age: 21
```

Changing Separator

- Use sep parameter to change the separator (default = space).

```
print("2025", "09", "11", sep="-")
```

Output:

```
2025-09-11
```

Changing End Character

- Use end parameter to change what happens at the end (default = newline).

```
print("Hello", end=" ")
print("World")
```

Output:

```
Hello World
```

String Formatting in Output

1. **f-strings (recommended, Python 3.6+)**

```
name = "Alice"
age = 21
print(f"My name is {name} and I am {age} years old.")
```

Output:

```
My name is Alice and I am 21 years old.
```

2. **str.format()**

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

3. **Old-style % formatting**

```
print("My name is %s and I am %d years old." % (name, age))
```

1.6.3 Real-Life Example: User Registration

```
username = input("Enter username: ")
age = int(input("Enter age: "))
email = input("Enter email: ")

print("\n--- User Details ---")
print(f"Username: {username}")
print(f"Age: {age}")
print(f"Email: {email}")
```

Output:

```
Enter username: john_doe
Enter age: 22
Enter email: john@example.com

--- User Details ---
Username: john_doe
Age: 22
Email: john@example.com
```

Key Takeaways

- **input()** always returns string → convert with int(), float() if needed.
- **print()** displays output, with options: sep, end, and string formatting.
- Use **f-strings** for cleaner, modern formatting.
- Input/Output is the foundation of interactive programs.

1.7 Type Conversion

1.7.1 What is Type Conversion?

Type conversion means **changing the data type of a value** into another type.

In Python, type conversion is mainly of two types:

1. **Implicit Type Conversion (Type Casting done by Python automatically)**
2. **Explicit Type Conversion (Manually done by the programmer using functions)**

1.7.2 Implicit Type Conversion

- Also known as **Type Casting / Type Promotion**.
- Python **automatically converts smaller data types into larger data types** to avoid data loss.

Example:

```
num_int = 10         # int
num_float = 2.5      # float

result = num_int + num_float   # int + float → float
print(result)                   # 12.5
print(type(result))             # <class 'float'>
```

👉 Here, Python **converted int to float** automatically.

1.7.3 Explicit Type Conversion

- Also called **Type Casting**.
- Done by using built-in functions: int(), float(), str(), bool(), etc.

Examples:

```
# Convert float to int
x = int(3.9)
```

```
print(x)          # 3

# Convert int to float
y = float(7)
print(y)          # 7.0

# Convert number to string
z = str(123)
print(z)          # '123'
print(type(z))    # <class 'str'>

# Convert string to int
s = int("42")
print(s + 8)      # 50
```

👉 Be careful: converting invalid strings will raise an error:

```
int("hello")    # ❌ ValueError
```

1.7.4 Common Type Conversion Functions

Function	Description	Example	Result
int(x)	Converts to integer	int(3.7)	3
float(x)	Converts to float	float(5)	5.0
str(x)	Converts to string	str(100)	100'
bool(x)	Converts to Boolean	bool(0)	FALSE
list(x)	Converts to list	list("abc")	['a', 'b', 'c']
tuple(x)	Converts to tuple	tuple([1,2,3])	(1, 2, 3)
set(x)	Converts to set (unique values)	set([1,1,2])	{1, 2}

1.7.5 Real-Life Example

```
# Shopping cart example
item_price = float(input("Enter item price: "))
quantity = int(input("Enter quantity: "))
total = item_price * quantity
```

```
print("\n--- Bill ---")
print("Price per item:", item_price)
print("Quantity:", quantity)
print("Total amount:", str(total) + " INR")
```

Output:

```
Enter item price: 49.99
Enter quantity: 3

--- Bill ---
Price per item: 49.99
Quantity: 3
Total amount: 149.97 INR
```

Key Takeaways

- **Implicit Conversion** → Python handles automatically (safe conversions).
- **Explicit Conversion** → Use functions like int(), float(), str().
- Always ensure the value is **valid for conversion**, else errors occur.
- Conversions are essential for **mathematical operations, string manipulations, and data handling**.

1.8 Operators in Python

1.8.1 What are Operators?

- Operators are **symbols or keywords** used to perform operations on variables and values.
- Example: +, -, *, /, ==, and, is, in etc.

👉 In Python, operators are classified into **seven main types**:

1. Arithmetic Operators
2. Relational (Comparison) Operators
3. Logical Operators
4. Assignment Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators

1.8.2 Arithmetic Operators

Used for basic mathematical calculations.

Operator	Description	Example	Result
+	Addition	10 + 5	15
-	Subtraction	10 - 3	7
*	Multiplication	4 * 3	12
/	Division (float result)	10 / 3	3.333...
//	Floor Division (integer result)	10 // 3	3
%	Modulus (remainder)	10 % 3	1
**	Exponent (power)	2 ** 3	8

Example:

```
a, b = 10, 3
print(a + b, a - b, a * b, a / b, a // b, a % b, a ** b)
```

1.8.3 Relational (Comparison) Operators

Used to compare values → result is always **True/False**.

Operator	Description	Example	Result
	Equal to	5 == 5	TRUE
!=	Not equal to	5 != 3	TRUE
>	Greater than	7 > 4	TRUE
<	Less than	2 < 8	TRUE
>=	Greater or equal	5 >= 5	TRUE
<=	Less or equal	3 <= 4	TRUE

Example:

```
x, y = 7, 10
print(x == y, x != y, x > y, x < y, x >= y, x <= y)
```

1.8.4 Logical Operators

Used to combine conditional statements.

Operator	Description	Example	Result
and	True if **both** are true	(x > 5 and y < 15)	TRUE

or	True if **at least one** is true	(x < 5 or y < 15)	TRUE
not	Reverses the condition	not(x > 5)	FALSE

Example:

```
x, y = 8, 12
print(x > 5 and y < 15)  # True
print(x > 10 or y < 15)  # True
print(not(x > 5))        # False
```

1.8.5 Assignment Operators

Used to assign values to variables (with optional operation).

Operator	Example	Equivalent To
=	x = 5	Assigns 5
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 3	x = x / 3
//=	x //= 2	x = x // 2
%=	x %= 2	x = x % 2
**=	x **= 3	x = x ** 3

Example:

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

1.8.6 Bitwise Operators

Work at the **binary (bit) level**.

Operator	Description	Example (x=6 (110), y=3 (011))	Result
&	AND	x & y	2 (010)
`	`	OR	`x
^	XOR	x ^ y	5 (101)
~	NOT	~x	-7
<<	Left Shift	x << 1	12 (1100)
>>	Right Shift	x >> 1	3 (011)

Example:

```
x, y = 6, 3
print(x & y, x | y, x ^ y, ~x, x << 1, x >> 1)
```

1.8.7 Membership Operators

Check whether a value exists inside a sequence (string, list, tuple, set, dict).

Operator	Example	Result
in	"a" in "apple"	TRUE
not in	"x" not in "apple"	TRUE

Example:

```
fruits = ["apple", "banana", "cherry"]
print("apple" in fruits)      # True
print("grape" not in fruits) # True
```

1.8.8 Identity Operators

Compare whether **two objects refer to the same memory location**.

Operator	Example	Result
is	x is y	True if same object
is not	x is not y	True if not same object

Example:

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a is b)       # True (same object)
print(a is c)       # False (different objects, even if same values)
print(a == c)       # True (values are same)
```

1.8.9 Real-Life Example: Simple Calculator

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
```

```
print("\n--- Operations ---")
print("Addition:", num1 + num2)
print("Subtraction:", num1 - num2)
print("Multiplication:", num1 * num2)
print("Division:", num1 / num2)
print("Modulus:", num1 % num2)
print("Exponent:", num1 ** num2)
```

Key Takeaways

- Python provides **7 types of operators**.
- Arithmetic → math, Relational → comparison, Logical → decision making.
- Assignment operators simplify updating values.
- Bitwise operators work at **binary level**.
- Membership checks data inside sequences.
- Identity checks **object references** in memory.

1.9 Control Flow in Python

1.9.1 What is Control Flow?

- Control flow refers to the **order in which statements are executed** in a program.
- By default, Python executes statements **sequentially (top to bottom)**.
- Using **decision-making statements** and **loops**, we can change this flow.

Control flow has two main parts:

1. **Decision Making (if, if-else, nested if, match-case)**
2. **Loops (for, while, else in loops, loop control statements)**

1.9.2 Decision-Making Statements

(a) if Statement

Used to execute a block of code **only if a condition is true**.

Syntax:

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

Example:

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

(b) if-else Statement

Used when we need an **alternative block of code** if the condition is false.

Syntax:

```
if condition:
    # true block
else:
    # false block
```

Example:

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

(c) if-elif-else Statement

Used when there are **multiple conditions**.

Syntax:

```
if condition1:
    # block1
elif condition2:
    # block2
else:
    # block3
```

Example:

```
marks = 72
if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
elif marks >= 50:
    print("Grade C")
else:
    print("Fail")
```

(d) Nested if

An if inside another if.

Example:

```
x = 20
if x > 10:
    if x < 30:
        print("x is between 10 and 30")
```

(e) match-case (Python 3.10+)

Similar to switch-case in other languages.

Syntax:

```
match variable:
    case value1:
        # code block
    case value2:
        # code block
    case _:
        # default case
```

Example:

```
day = 3
match day:
    case 1:
        print("Monday")
    case 2:
        print("Tuesday")
    case 3:
        print("Wednesday")
    case _:
        print("Invalid day")
```

1.9.3 Loops in Python

(a) for Loop

Used for iterating over a sequence (list, string, range, etc.).

Syntax:

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

Example:

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

Output:

```
1 2 3 4 5
```

(b) while Loop

Repeats a block of code **as long as the condition is true**.

Syntax:

```
while condition:
    # code block
```

Example:

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

(c) else with Loops

The else block executes **when the loop finishes normally** (not terminated by break).

Example:

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

Output:

```
0
```

```
1
2
Loop finished
```

(d) Loop Control Statements

1. **break** → exits the loop immediately.

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output: 0 1 2

2. **continue** → skips current iteration and continues.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Output: 0 1 3 4

3. **pass** → placeholder (does nothing).

```
for i in range(5):
    pass  # future implementation
```

1.9.4 Real-Life Examples

Example 1: ATM Withdrawal

```
balance = 5000
amount = int(input("Enter amount to withdraw: "))

if amount <= balance:
    balance -= amount
        print("Withdrawal successful. Remaining balance:",
balance)
else:
    print("Insufficient balance.")
```

Example 2: Student Attendance Check

```
attendance = 85

if attendance >= 75:
    print("Allowed to sit for exam")
else:
    print("Not allowed")
```

Example 3: Multiplication Table using Loop

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

for i in range(1, 11):
    print(f"{num} x {i} = {num*i}")
```

Key Takeaways

- Control flow decides **which code runs and how many times**.
- Use **if, if-else, if-elif-else, match-case** for decisions.
- Use **for loops** for sequences, **while loops** for conditions.
- break, continue, and pass give more control in loops.

1.10 Loop Control Statements in Python

1.10.1 What are Loop Control Statements?

- Python provides special keywords to **control the behavior of loops**.
- These allow us to **skip iterations, exit loops early, or handle unused blocks**.
- The main loop control statements are:
    1. break

2. continue
3. pass
4. else with loops

1.10.2 The break Statement

- Immediately **terminates the loop** (for or while).
- Control moves to the **first statement after the loop**.

Example 1: Stop at a number

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

Output:

```
1 2 3 4
```

Real-Life Example: ATM PIN Check

```
correct_pin = "1234"
for attempt in range(3):
    pin = input("Enter PIN: ")
    if pin == correct_pin:
        print("Access Granted ✅")
        break
else:
    print("Card Blocked ❌")
```

1.10.3 The continue Statement

- Skips the **current iteration** and moves to the **next iteration**.

Example 1: Skip even numbers

```
for i in range(1, 6):
    if i % 2 == 0:
        continue
    print(i, end=" ")
```

Output:

```
1 3 5
```

Real-Life Example: Filtering Students

```
students = ["Anu", "", "Rahul", "", "Meera"]

for name in students:
    if name == "":
        continue
    print("Present:", name)
```

1.10.4 The pass Statement

- A **do-nothing placeholder**.
- Used when a statement is **syntactically required** but you don't want any code to run yet.

Example 1: Future Code Placeholder

```
for i in range(5):
    pass   # To be implemented later
```

Example 2: Empty Function

```
def todo_feature():
    pass
```

1.10.5 The else with Loops

- Executes **only if the loop completes normally** (i.e., not interrupted by break).

Example 1: Normal loop

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

Output:

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

Example 2: Loop with break

```
for i in range(3):
    if i == 1:
        break
    print(i)
else:
    print("Loop finished successfully")
```

Output:

```
0
```

👉 The else block was skipped because loop ended with break.

1.10.6 Nested Loops with Control Statements

Loop control works inside **nested loops** too.

Example: Multiplication table (skip odd numbers)

```
for i in range(1, 6):
    if i % 2 != 0:
        continue
    for j in range(1, 6):
        print(f"{i} x {j} = {i*j}")
    print("---")
```

Key Takeaways

- **break** → exit loop immediately.
- **continue** → skip current iteration, move to next.
- **pass** → placeholder (do nothing).
- **else with loop** → executes only if loop finishes normally.
- Helps build **flexible and controlled logic** inside loops.