Python

**Day2**:

**Advanced Rules for Variables in Python**

1. **Naming Rules:**
   * Can contain letters (a–z, A–Z), digits (0–9), and underscores \_.
   * Cannot start with a digit.
   * Cannot be a Python **keyword** or built-in function name.
   * Case-sensitive (age ≠ Age).( Python, C / C++,Java, JavaScript)but not case sensitive(SQL, BASIC)
2. **Dynamic Typing:**
   * Python variables **don’t require type declaration**.
   * The type is determined automatically based on the value.

x = 10 # int

x = "Ram" # str (type changes dynamically)

1. **Multiple Assignment:**
   * Python allows assigning multiple variables in one line:

a, b, c = 1, 2, 3

**Valid Examples:**

student\_name = "Ram"

\_score = 95

totalMarks123 = 100

1. **Invalid Examples:**

1st\_name = "Ram" # starts with digit ❌

for = 10 # keyword ❌

1. **Memory Reference:**
   * Variables in Python are **references to objects** in memory.
   * Changing the value of a variable points it to a new object.
2. **Global & Local Scope:**
   * Variables can be **global** (accessible anywhere) or **local** (inside a function only).
   * Sure Ram! Here’s a simple explanation of **why we use different naming cases** in Python and programming:

**🔹 Why We Use Different Naming Cases**

1. **Readability** ✅

* Naming conventions make code **easy to read and understand**.
* Example: student\_name = "Ram" # easy to read (snake\_case)

1. **Consistency** 🎯

* Using a standard case across a project keeps code **consistent**.
* Easier for **team collaboration**.

1. **Indicates Purpose / Type** 🔍

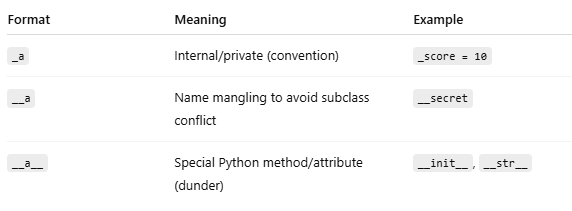
* Different cases help programmers **know what the variable represents**:
  + snake\_case → regular variable or function
  + PascalCase → class name
  + UPPER\_CASE → constant value

1. **Avoids Errors** ⚠️

* Helps avoid naming conflicts and mistakes in large programs.

✅ **Summary Table:**

| **Case** | **Use in Python** | **Example** |
| --- | --- | --- |
| Snake Case (All lowercase letters, words separated by \_) | Variables, functions | student\_name |
| Camel Case(First word lowercase, following words start with uppercase, no spaces) | Rare in Python, common in JS/Java(Variables and functions in some other languages like Java, JavaScript) | studentName |
| Pascal Case(Every word starts with uppercase, no spaces) | Classes | StudentName |
| Kebab Case(not common in Python, more in URLs/filenames) | URLs, filenames | student-name |
| Upper Case(All letters uppercase, words separated by \_) | Constants | PI, MAX\_VALUE |



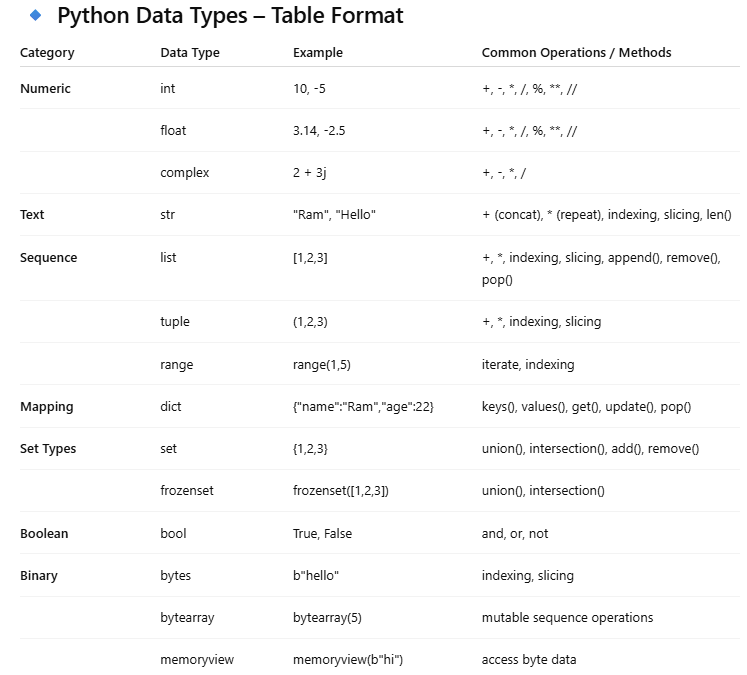
Paxadi ramro snaga details ma xa

Mainly 36 reserver word/keyword

Code : Import keyword

Print(keyword.kwlist)

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



**🔹 Number Systems in Python**

Python supports **different number systems** for integers.

| **System** | **Prefix in Python** | **Example** | **Base** |
| --- | --- | --- | --- |
| **Binary** | 0b or 0B | 0b1010 | 2 |
| **Decimal** | No prefix | 10 | 10 |
| **Octal** | 0o or 0O | 0o12 | 8 |
| **Hexadecimal** | 0x or 0X | 0xA | 16 |

**🔹 Examples in Python**

# Binary

x = 0b1010

print(x) # 10 (decimal)

# Decimal

y = 10

print(y) # 10

# Octal

z = 0o12

print(z) # 10 (decimal)

# Hexadecimal

h = 0xA

print(h) # 10 (decimal)

**🔹 Key Points**

1. Python **automatically converts** to decimal when doing calculations.
2. Use **bin(), oct(), hex()** to convert decimal numbers to other bases:

n = 10

print(bin(n)) # 0b1010

print(oct(n)) # 0o12

print(hex(n)) # 0xa

**🔹 id() in Python**

**Definition:**

* The **id() function** in Python returns the **unique identity (address) of an object** in memory.
* Every object in Python has a **unique ID**, which is its **memory location**.

**Day3:** Complex number data type:

**1. Syntax**

z = a + bj

* a → Real part
* b → Imaginary part
* j → Imaginary unit (√-1)

**2. Example**

z1 = 3 + 4j

z2 = 1 - 2j

print(z1) # (3+4j)

print(type(z1)) # <class 'complex'>

**3. Accessing Parts**

* real → real part
* imag → imaginary part

z = 3 + 4j

print(z.real) # 3.0

print(z.imag) # 4.0

**4. Arithmetic Operations**

z1 = 3 + 4j

z2 = 1 + 2j

print(z1 + z2) # (4+6j)

print(z1 - z2) # (2+2j)

print(z1 \* z2) # (-5+10j)

print(z1 / z2) # (2.2-0.4j)

**5. Built-in Functions**

import cmath

z = 3 + 4j

print(abs(z)) # 5.0 (magnitude)

print(cmath.phase(z)) # 0.927 radians (angle θ)

print(cmath.polar(z)) # (5.0, 0.927) → (r, θ)

**6. Create using complex() function**

z = complex(5, -2)

print(z) # (5-2j)

**✅ Boolean Data Type in Python**

**1. What is Boolean?**

* **Boolean (bool)** is a built-in data type in Python that can hold only **two values**:
  + True(1)(aagadaiko letter jahile ni capital hunu parxa hai bro)
  + False(0)
* It is mainly used in **logical conditions**, **comparisons**, and **decision making**.

**2. Example**

x = True

y = False

print(x) # True

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

print(x+y) #output is 1 bro

**3. Boolean from Comparisons**

Booleans are often produced by **comparison operators**:

a = 10

b = 20

print(a > b) # False

print(a < b) # True

print(a == 10) # True

**4. Boolean Operators**

Boolean values can be combined using **logical operators**:

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| 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 value | not True → False |

Example:

x = True

y = False

print(x and y) # False

print(x or y) # True

print(not x) # False

**5. Boolean Conversion**

You can use the bool() function to convert other data types into Boolean.

👉 Rule:

* **Falsy values → False**: 0, 0.0, "", [], {}, set(), None
* **Truthy values → True**: everything else

print(bool(0)) # False

print(bool(42)) # True

print(bool("")) # False

print(bool("Hello")) # True

print(bool([])) # False

print(bool([1,2,3])) # True

**6. Usage in Conditions**

is\_logged\_in = True

if is\_logged\_in:

print("Welcome User!")

else:

print("Please login")

**📝 String Data Type in Python**

-Not char in python( all string ho bro)

* A **string** in Python is a sequence of **characters** enclosed in **single quotes '...'**, **double quotes "..."**, or **triple quotes '''...''' or """..."""**.
* Strings are **immutable** (once created, they cannot be changed).

**2. Creating Strings**

str1 = 'Hello'

str2 = "World"

str3 = '''This is

a multi-line

string.''' #multiline line kolagi chai triple “”” or’’’ use bro

print(str1) # Hello

print(str2) # World

print(str3)

**3. Accessing Characters**

Strings are like arrays of characters → you can access them using **indexing**.

* Index starts from 0.
* Negative index starts from the end.

s = "Python"

print(s[0]) # P

print(s[3]) # h

print(s[-1]) # n (last character)

**4. String Slicing**

You can extract a portion (substring) using slicing:

s = "Python"

print(s[0:4]) # Pyth (from index 0 to 3)

print(s[:3]) # Pyt (from start to 2)

print(s[2:]) # thon (from 2 to end)

print(s[::-1]) # nohtyP (reversed string)

**5. String Operations**

a = "Hello"

b = "World"

# Concatenation

print(a + " " + b) # Hello World

# Repetition

print(a \* 3) # HelloHelloHello

# Membership

print("H" in a) # True in → checks if a substring exists inside a string.

print("z" not in a) # True not in → checks if a substring does not exist.

**6. Useful String Methods**

Python provides many built-in string methods:

s = " Python Programming "

print(s.upper()) # PYTHON PROGRAMMING

print(s.lower()) # python programming

print(s.strip()) # "Python Programming" (removes spaces)

print(s.replace("Python", "Java")) # Java Programming

print(s.split()) # ['Python', 'Programming']

print(s.startswith("Py")) # True

print(s.endswith("ing")) # True

print(len(s)) # 22 (length)

**7. String Formatting**

**f-strings (modern way, Python 3.6+)**

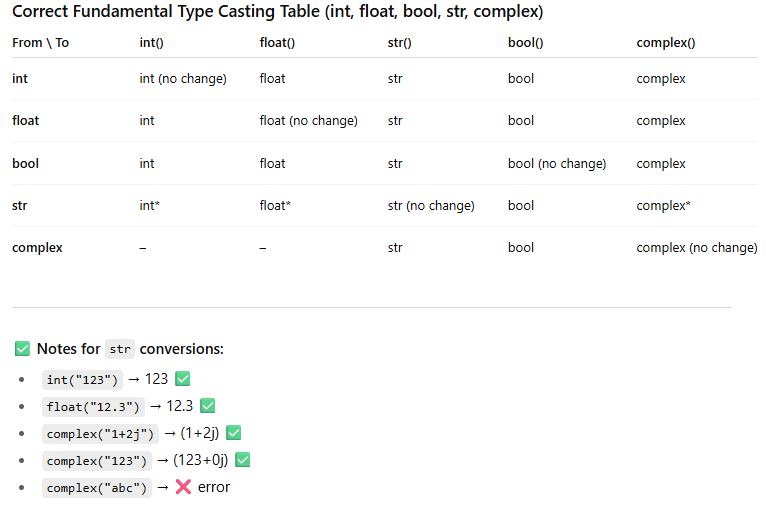
name = "Ram"

age = 21

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

**format() method**

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

Type casting:

**Fundamental Data Types**

* int, float, complex, bool, str

**Immutability**

* All fundamental data types are **immutable**.
* Once created, the object **cannot be changed**.
* Any modification creates a **new object**.

**Who destroys the old object?**

👉 **Python’s Garbage Collector (GC)** automatically destroys objects that are **no longer referenced**.

That means:

* If no variable points to an object, it becomes **unreachable**.
* Then the Garbage Collector **frees the memory**.

✅ Example:

x = 10

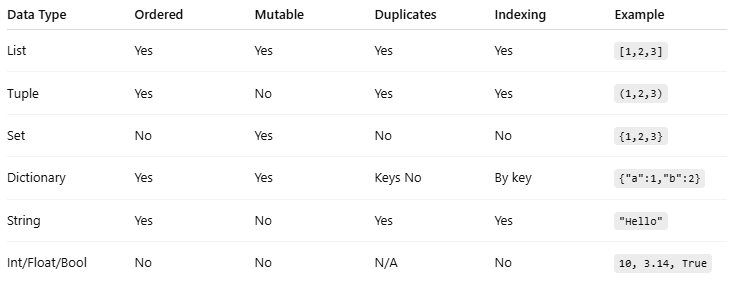
print(id(x)) # address of 10

x = x + 5 # now x refers to new object 15

print(id(x)) # different address

# object 10 has no reference now → Garbage Collector will destroy it

**Day 4**



**Python List Data Type**

**Definition:**

A **list** in Python is an **ordered, mutable collection** of items.

* **Ordered**: Elements have a defined order, and you can access them by index.
* **Mutable**: You can **change, add, or remove elements** after the list is created.
* **Heterogeneous**: A list can contain **different data types** (int, float, string, other lists, etc.).

**Syntax:**

my\_list = [element1, element2, element3, ...]

**Example:**

fruits = ["apple", "banana", "cherry"]

* **Ordered ≠ unchangeable order**.
  + Being ordered just means **Python remembers the sequence of elements**.
  + The **order can be changed** by operations like insert(), append(), sort(), reverse().

**Key Features of Lists:**

1. **Indexing and Slicing**
2. fruits[0] # "apple" -> first element
3. fruits[-1] # "cherry" -> last element
4. fruits[0:2] # ["apple", "banana"] -> slicing
5. **Mutable**
6. fruits[1] = "orange" # change second element
7. **Heterogeneous**
8. mixed = [1, "apple", 3.5]
9. **Dynamic Size**
   * Can add or remove elements anytime.

**Common List Operations in Python**

**1. Adding Elements**

**a) append()**

* Adds a single element at the **end** of the list.
* **Syntax:** list.append(element)
* **Example:**

fruits = ["apple", "banana"]

fruits.append("cherry")

print(fruits) # Output: ['apple', 'banana', 'cherry']

**b) extend()**

* Adds **multiple elements** from another list (or iterable) at the **end**.
* **Syntax:** list.extend(iterable)
* **Example:**

fruits = ["apple", "banana"]

fruits.extend(["cherry", "mango"])

print(fruits) # Output: ['apple', 'banana', 'cherry', 'mango']

**c) insert()**

* Inserts an element at a **specific index**.
* **Syntax:** list.insert(index, element)
* **Example:**

fruits = ["apple", "banana"]

fruits.insert(1, "cherry")

print(fruits) # Output: ['apple', 'cherry', 'banana']

**2. Removing Elements**

**a) remove()**

* Removes **first occurrence** of a specified element.
* **Syntax:** list.remove(element)
* **Example:**

fruits = ["apple", "banana", "cherry"]

fruits.remove("banana")

print(fruits) # Output: ['apple', 'cherry']

**b) pop()**

* Removes element at a **specific index** and **returns it**. Default is last element.
* **Syntax:** list.pop(index)
* **Example:**

fruits = ["apple", "banana", "cherry"]

last = fruits.pop()

print(last) # Output: cherry

print(fruits) # Output: ['apple', 'banana']

**c) clear()**

* Removes **all elements** from the list.
* **Syntax:** list.clear()
* **Example:**

fruits = ["apple", "banana", "cherry"]

fruits.clear()

print(fruits) # Output: []

**3. Other Useful Operations**

**a) len()**

* Returns the **number of elements** in the list.
* **Example:**

fruits = ["apple", "banana", "cherry"]

print(len(fruits)) # Output: 3

**b) sort()**

* Sorts the list **in ascending order** (by default).
* **Syntax:** list.sort(key=None, reverse=False)
* **Example:**

numbers = [3, 1, 4, 2]

numbers.sort()

print(numbers) # Output: [1, 2, 3, 4]

numbers.sort(reverse=True)

print(numbers) # Output: [4, 3, 2, 1]

**c) reverse()**

* Reverses the **order of elements** in the list.
* **Example:**

fruits = ["apple", "banana", "cherry"]

fruits.reverse()

print(fruits) # Output: ['cherry', 'banana', 'apple']

**d) copy()**

* Returns a **shallow copy** of the list.
* **Example:**

fruits = ["apple", "banana"]

new\_fruits = fruits.copy()

print(new\_fruits) # Output: ['apple', 'banana']

**e) count()**

* Returns the **number of times an element occurs**.
* **Example:**

numbers = [1, 2, 2, 3]

print(numbers.count(2)) # Output: 2

**f) index()**

* Returns the **index of first occurrence** of an element.
* **Example:**

fruits = ["apple", "banana", "cherry"]

print(fruits.index("banana")) # Output: 1



**Advanced Python Tuples –**

**1. Definition Recap**

👉 Tuple is an **ordered, immutable, heterogeneous collection**.

* **Ordered:** Index system cha (0, 1, 2, …).
* **Immutable:** Once tuple create garyo, elements change, add, remove garna mildaina.
* **Heterogeneous:** Different datatypes store garna sakcha.
* **Syntax:**

my\_tuple = (1, "apple", 3.5, True)

**2. Nested Tuples**

Tuple bhitra **tuple** or **list** rakhna milcha.

nested = (1, 2, (3, 4), [5, 6])

print(nested[2]) # (3, 4)

print(nested[3][1]) # 6

⚡ Note: Tuple immutable ho, tara inside list chai mutable huncha (change garna milcha).

**3. Tuple Unpacking (Destructuring)**

Multiple elements lai directly variables ma unpack garna milcha.

point = (10, 20, 30)

x, y, z = point

print(x, y, z) # 10 20 30

**Extended Unpacking (with \*)**

numbers = (1, 2, 3, 4, 5)

a, \*b, c = numbers

print(a) # 1

print(b) # [2, 3, 4]

print(c) # 5

**4. Tuple Operations**

| **Operation** | **Example** | **Output** |
| --- | --- | --- |
| Concatenation (+) | (1,2) + (3,4) | (1,2,3,4) |
| Repetition (\*) | (1,2) \* 2 | (1,2,1,2) |
| Membership (in) | 2 in (1,2,3) | True |
| Indexing | t[0] | First element |
| Slicing | t[1:3] | Sub-tuple |

**5. Tuple Methods (Only 2)**

Tuples immutable huda list jasto methods hudaina. Available methods:

1. count(x) → number of times x occurs
2. index(x) → first index of x

t = (1, 2, 2, 3)

print(t.count(2)) # 2

print(t.index(3)) # 3

**6. Single Element Tuple**

Ek element ko tuple banaunda **comma mandatory** ho.

single = (5,)

print(type(single)) # <class 'tuple'>

not\_tuple = (5)

print(type(not\_tuple)) # <class 'int'>

**7. Immutability Deep Dive**

* Tuple itself immutable → t[1] = 99 → ❌ Error
* But inside mutable element (like list) change garna milcha:

t = (1, [2, 3], 4)

t[1][0] = 99

print(t) # (1, [99, 3], 4)

👉 Tuple pointer fixed huncha, tara inside list modify garna milcha.

**8. Tuple as Dictionary Keys**

* Immutable huda tuple lai **dictionary key** banauna milcha.
* List cannot be a key (mutable huney le).

my\_dict = {(1,2): "value"}

print(my\_dict[(1,2)]) # value

**9. Tuple Packing & Unpacking**

* **Packing:** Multiple values → tuple automatically

t = 1, 2, 3 # Packed tuple

* **Unpacking:** Tuple → multiple variables

a, b, c = t

print(a, b, c) # 1 2 3

**10. Named Tuples (Advanced Use)**

collections.namedtuple → Tuple with named fields.

* More readable than normal tuple.

from collections import namedtuple

Point = namedtuple("Point", ["x", "y"])

p = Point(10, 20)

print(p.x, p.y) # 10 20

**11. Use Cases of Tuples**

✔ Store fixed data (like coordinates, days of week)  
✔ As dictionary keys (immutable requirement)  
✔ Returning multiple values from function  
✔ Safer than lists (no accidental modification)  
✔ More memory efficient & slightly faster

**12. Difference: List vs Tuple**

| **Feature** | **List** | **Tuple** |
| --- | --- | --- |
| Mutability | ✅ Mutable | ❌ Immutable |
| Syntax | [] | () |
| Methods | Many (append, extend, pop, …) | Only count, index |
| Performance | Slower | Faster |
| Use Case | Dynamic, frequently changing data | Fixed, constant data |

**Set in Python**

**1. Definition**

👉 **Set** is a **built-in data type** in Python, which is:

* **Unordered** → elements ko order fixed hudaina.
* **Mutable** → elements add/remove garna milcha.
* **Unique elements only** → duplicates automatically remove huncha.

**Syntax:**

s = {1, 2, 3, 4}

print(s) # {1, 2, 3, 4}

Or use constructor:

s = set([1, 2, 2, 3])

print(s) # {1, 2, 3}

**2. Properties of Set**

* ✅ No duplicates
* ✅ Unordered (no indexing, no slicing)
* ✅ Mutable (change garna milcha)
* ✅ Can only contain **immutable (hashable) elements** (numbers, strings, tuples)
* ❌ Cannot contain mutable elements (list, dict, set)

**3. Creating Sets**

# Empty set

s = set() # ✅ Correct way

s2 = {} # ❌ This creates a dictionary, not a set

# From string

s = set("apple")

print(s) # {'a', 'l', 'e', 'p'}

**4. Common Set Methods**

**Add Elements**

* add(x) → Add single element
* update(iterable) → Add multiple elements

s = {1, 2}

s.add(3)

print(s) # {1, 2, 3}

s.update([4, 5])

print(s) # {1, 2, 3, 4, 5}

**Remove Elements**

* remove(x) → Removes element (error if not found)
* discard(x) → Removes element (no error if not found)
* pop() → Removes random element
* clear() → Removes all elements

s = {1, 2, 3}

s.remove(2)

print(s) # {1, 3}

s.discard(5) # No error even if not found

s.pop() # Removes random element

s.clear() # Set becomes empty

**5. Set Operations**

**Union (| or .union())**

a = {1, 2, 3}

b = {3, 4, 5}

print(a | b) # {1, 2, 3, 4, 5}

print(a.union(b)) # same result

**Intersection (& or .intersection())**

print(a & b) # {3}

**Difference (- or .difference())**

print(a - b) # {1, 2}

**Symmetric Difference (^ or .symmetric\_difference())**

print(a ^ b) # {1, 2, 4, 5}

**6. Other Useful Methods**

* len(s) → number of elements
* copy() → shallow copy of set
* issubset(other) → checks if set is subset
* issuperset(other) → checks if set is superset
* isdisjoint(other) → True if sets have no common elements

a = {1, 2}

b = {1, 2, 3}

print(a.issubset(b)) # True

print(b.issuperset(a)) # True

print(a.isdisjoint({4,5})) # True

**8. Limitation**

* ❌ No indexing or slicing (s[0] → error)
* ❌ Cannot store mutable elements (list, dict, set)

s = {1, [2,3]} # ❌ Error: unhashable type 'list'

**9. Use Cases**

* Removing duplicates from list
* Performing mathematical set operations (union, intersection, …)
* Membership testing (in is faster in sets than lists)

**10. Example: Removing Duplicates**

nums = [1, 2, 2, 3, 4, 4, 5]

unique\_nums = set(nums)

print(unique\_nums) # {1, 2, 3, 4, 5}

**11. Difference: Set vs Frozen Set**

| **Feature** | **Set** | **Frozen Set** |
| --- | --- | --- |
| Mutability | ✅ Mutable | ❌ Immutable |
| Methods | add(), remove(), pop()… | Only read-only (union, intersection, …) |
| Hashable | ❌ No | ✅ Yes |
| Dictionary Key | ❌ No | ✅ Yes |

Frozen set

🔹 Frozenset in Python (Advanced)

1. Basic Concept

* A normal Python set → mutable (we can add/remove elements).
* A frozenset → immutable (once created, elements cannot be changed).
* frozenset is hashable → that means it can be used as a dictionary key or an element inside another set.

# Normal set

s = {1, 2, 3}

s.add(4) # Works

print(s) # {1, 2, 3, 4}

# Frozenset

fs = frozenset([1, 2, 3])

# fs.add(4) # ❌ Error: 'frozenset' object has no attribute 'add'

print(fs) # frozenset({1, 2, 3})

2. Creating a Frozenset

fs1 = frozenset([1, 2, 3, 4])

fs2 = frozenset("hello")

print(fs1) # frozenset({1, 2, 3, 4})

print(fs2) # frozenset({'h', 'e', 'l', 'o'}) (duplicates removed automatically)

3. Allowed Operations

Even though frozenset is immutable, we can still perform set operations (they return a new frozenset):

A = frozenset([1, 2, 3, 4])

B = frozenset([3, 4, 5, 6])

print(A.union(B)) # frozenset({1, 2, 3, 4, 5, 6})

print(A.intersection(B)) # frozenset({3, 4})

print(A.difference(B)) # frozenset({1, 2})

print(A.symmetric\_difference(B)) # frozenset({1, 2, 5, 6})

print(A.isdisjoint(B)) # False

print(A.issubset(B)) # False

print(A.issuperset(B)) # False

4. Why is Frozenset Useful?

Advanced use cases:

1. Dictionary Key

d = {

frozenset([1, 2]): "A",

frozenset([3, 4]): "B"

}

print(d[frozenset([1, 2])]) # A

1. Set of Sets Problem  
   A normal set cannot hold another set (because sets are mutable). But a frozenset can be stored inside a set:

s = {frozenset([1, 2]), frozenset([3, 4])}

print(s) # {frozenset({1, 2}), frozenset({3, 4})}

1. Data Integrity / Immutability  
   If you want to make sure that data never changes (for example, fixed configuration sets), frozenset is a good choice.

5. Advanced Example – Graph Representation

We can use frozensets to store connections (edges) in a graph:

graph = {

"A": frozenset(["B", "C"]),

"B": frozenset(["A", "D"]),

"C": frozenset(["A", "D"]),

"D": frozenset(["B", "C"])

}

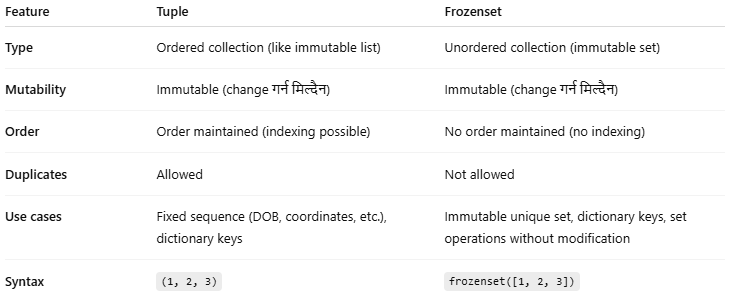
print(graph["A"]) # frozenset({'B', 'C'})

6. Performance Note

* Since frozenset is hashable → dictionary key lookups are fast.
* Because it is immutable → safe to use in multi-threaded programming (no accidental changes).

Summary (Key Points):

* frozenset = immutable version of set.
* Supports set operations but cannot be modified.
* Can be used as dictionary key or as an element of another set.
* Useful in graph theory, caching, functional programming, and when immutability is required.



**Dictionary**

* Dictionary = **key–value pair** data structure in Python.
* Key must be **immutable** (string, int, tuple, etc.).
* Value anything हुन सक्छ (list, dict, set, etc.).
* **Syntax**:
* my\_dict = {"name": "Ram", "age": 22, "city": "Kathmandu"}

**🔹 2. Accessing Elements**

print(my\_dict["name"]) # Ram

print(my\_dict.get("age")) # 22

print(my\_dict.get("gender", "Not Found")) # Not Found

**🔹 3. Updating & Adding**

my\_dict["age"] = 23 # update value

my\_dict["gender"] = "Male" # add new key-value

**🔹 4. Removing Elements**

my\_dict.pop("city") # remove key

my\_dict.popitem() # remove last inserted item

del my\_dict["name"] # delete key

my\_dict.clear() # empty dictionary

**🔹 5. Iteration**

for k in my\_dict.keys(): # keys

print(k)

for v in my\_dict.values(): # values

print(v)

for k, v in my\_dict.items(): # key + value

print(k, "=>", v)

**🔹 6. Dictionary Methods**

* get(key, default) → safely access
* setdefault(key, default) → insert if missing
* update(other\_dict) → merge dictionaries
* copy() → shallow copy

**🔹 7. Dictionary Comprehension**

squares = {x: x\*\*2 for x in range(5)}

print(squares) # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

**🔹 8. Nested Dictionary**

students = {

"ram": {"age": 22, "dept": "CSE"},

"shyam": {"age": 21, "dept": "ECE"}

}

print(students["ram"]["dept"]) # CSE

**🔹 9. Advanced (Python 3.9+ Merging)**

d1 = {"a": 1, "b": 2}

d2 = {"b": 3, "c": 4}

merged = d1 | d2

print(merged) # {'a': 1, 'b': 3, 'c': 4}

**🔹 10. Special from collections**

from collections import defaultdict, Counter

# defaultdict

d = defaultdict(int)

d["a"] += 1

print(d) # defaultdict(<class 'int'>, {'a': 1})

# Counter

data = ["apple", "banana", "apple"]

print(Counter(data)) # Counter({'apple': 2, 'banana': 1})

**1. range**

Think of range as **a sequence of numbers**.

range(start, stop, step)

* start → first number (default 0)
* stop → stop **before this number**
* step → how much to jump each time (default 1)

**Examples:**

r = range(5) # 0,1,2,3,4

r = range(2, 10) # 2,3,4,5,6,7,8,9

r = range(1, 10, 2) # 1,3,5,7,9

r = range(10, 0, -2) # 10,8,6,4,2

* Works **without making a full list** (saves memory)
* You can use it in loops:

for i in range(5):

print(i)

**2. bytes**

Think of bytes as **letters or numbers in computer language** (0–255 each).

* It’s like a string, but **for computers**, not people.

**Create bytes:**

b = b'hello' # from text

b = bytes([72,101,108,108,111]) # from numbers (ASCII)

b = bytes(5) # empty bytes: b'\x00\x00\x00\x00\x00'

**Use bytes:**

b = b'abc'

print(b[0]) # 97 (ASCII of 'a')

print(b[1:3]) # b'bc' (slice gives bytes)

**Convert bytes to string:**

s = b.decode('utf-8') # 'abc'

b2 = s.encode('utf-8') # b'abc'

* Can add or repeat:

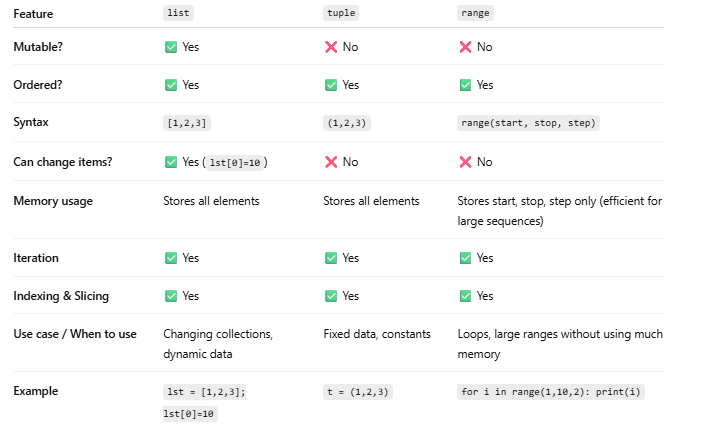
b1 = b'hi'

b2 = b'bye'

print(b1 + b2) # b'hibye'

print(b1\*3) # b'hihihi'





**Day 5:**

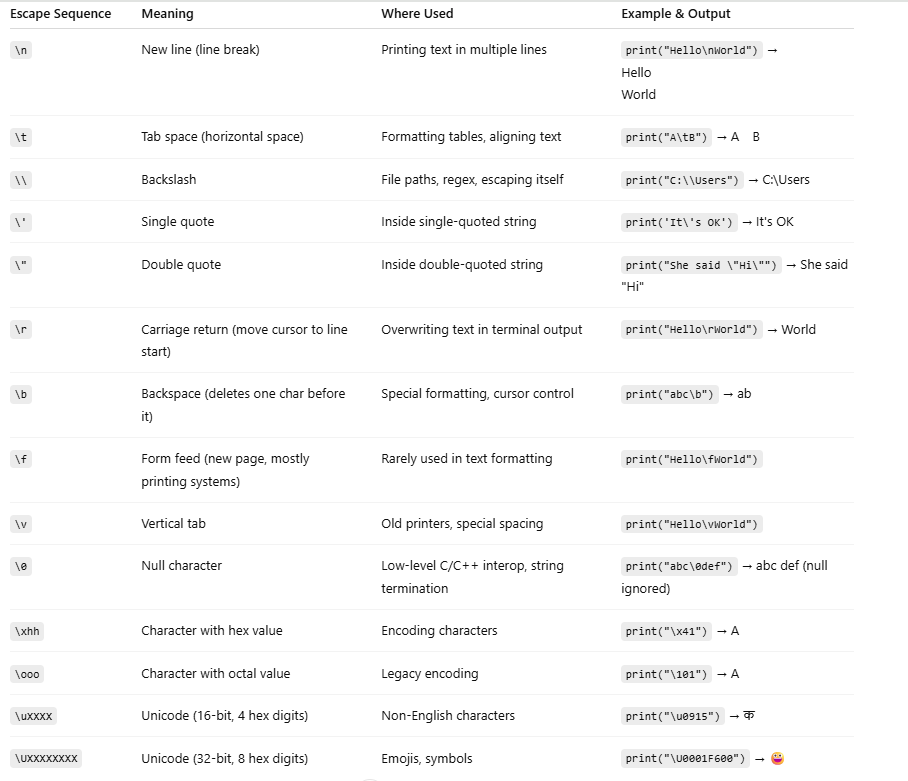
Python Escape Sequences

Escape sequences are special characters used in strings that cannot be typed directly or have special meanings. They start with a backslash \.

| Escape Sequence | Meaning / Use | Example |
| --- | --- | --- |
| \n | New line | print("Hello\nWorld") → Hello |
|  |  | World |
|  |  |  |
| \t | Tab (horizontal space) | print("A\tB") → A B |
| \\ | Backslash itself | print("C:\\Users") → C:\Users |
| \' | Single quote | print('It\'s OK') → It's OK |
| \" | Double quote | print("She said \"Hi\"") → She said "Hi" |
| \r | Carriage return (return to line start) | print("Hello\rWorld") → World |
| \b | Backspace | print("abc\b") → ab |
| \f | Form feed (new page, mostly for printers) | print("Hello\fWorld") |
| \v | Vertical tab | print("Hello\vWorld") |
| \0 | Null character | print("abc\0def") |
| \xhh | Hexadecimal value | print("\x41") → A |
| \ooo | Octal value | print("\101") → A |
| \uXXXX | Unicode (4 hex digits) | print("\u0915") → क |
| \UXXXXXXXX | Unicode (8 hex digits) | print("\U0001F600") → 😀 |

Easy way to remember:

* \n → new line
* \t → tab
* \\ → backslash
* \' / \" → quotes
* \u / \U → Unicode



**🐍 Python Operators**

**1. Arithmetic Operators**

Used for **mathematical operations**.

| **Operator** | **Description** | **Example** | **Output** |
| --- | --- | --- | --- |
| + | Addition | 10 + 5 | 15 |
| - | Subtraction | 10 - 3 | 7 |
| \* | Multiplication | 4 \* 3 | 12 |
| / | Division (float) | 10 / 4 | 2.5 |
| // | Floor Division (no decimals) | 10 // 4 | 2 |
| % | Modulus (remainder) | 10 % 3 | 1 |
| \*\* | Exponent (power) | 2 \*\* 3 | 8 |

✅ **Use when:** You need normal math operations in Python programs.

**2. Relational (Comparison) Operators**

👉 Used to **compare two values** (gives True or False).

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | 5 == 5 | True |
| != | Not equal | 5 != 3 | True |
| > | Greater than | 7 > 3 | True |
| < | Less than | 2 < 5 | True |
| >= | Greater or equal | 5 >= 5 | True |
| <= | Less or equal | 3 <= 5 | True |

✅ **Use when:** Making decisions in if, while, for loops.

**3. Logical Operators**

👉 Combine multiple conditions.

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| and | True if **both** true | (5 > 3 and 10 > 7) | True |
| or | True if **any one** true | (5 > 10 or 7 > 3) | True |
| not | Reverses the result | not(5 > 3) | False |

✅ **Use when:** Checking multiple conditions together.

**4. Bitwise Operators**

👉 Work at **binary level (0s and 1s)**.

Example:

a = 5 → 0101 (binary)

b = 3 → 0011

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| & | AND (1 if both 1) | 5 & 3 | 1 (0001) |
| ` | ` | OR (1 if any 1) | `5 |
| ^ | XOR (1 if different) | 5 ^ 3 | 6 (0110) |
| ~ | NOT (flip bits) | ~5 | -6 |
| << | Left shift (multiply by 2^n) | 5 << 1 | 10 |
| >> | Right shift (divide by 2^n) | 5 >> 1 | 2 |

✅ **Use when:** Doing **low-level programming, cryptography, compression, networking**.

**5. Assignment Operators**

👉 Assign values with operations.

| **Operator** | **Meaning** | **Example** | **Equivalent** |
| --- | --- | --- | --- |
| = | Assign | x = 10 | — |
| += | Add & assign | x += 3 | x = x + 3 |
| -= | Subtract & assign | x -= 2 | x = x - 2 |
| \*= | Multiply & assign | x \*= 4 | x = x \* 4 |
| /= | Divide & assign | x /= 2 | x = x / 2 |
| //= | Floor divide & assign | x //= 3 | x = x // 3 |
| %= | Modulus & assign | x %= 2 | x = x % 2 |
| \*\*= | Power & assign | x \*\*= 2 | x = x \*\* 2 |
| `&=, | =, ^=, <<=, >>=` | Bitwise assign | x &= 3 |

✅ **Use when:** Updating variables in loops, counters, accumulators.

**6. Equality Operators**

👉 (Often included inside relational operators, but separated in notes)

* == → checks **equal values**
* != → checks **not equal**

✅ **Use when:** Need exact equality check.

**7. Shift Operators**

👉 Part of bitwise operations.

* << → Left shift (move bits left → multiply by 2)
* >> → Right shift (move bits right → divide by 2)

x = 5 # 101

print(x << 1) # 10 (1010)

print(x >> 1) # 2 (10)

✅ **Use when:** Optimizing multiplication/division by powers of 2.

**8. Ternary Operator**

👉 One-line **if-else**.

**Syntax:**

value\_if\_true if condition else value\_if\_false

**Example:**

x = 10

result = "Even" if x % 2 == 0 else "Odd"

print(result) # Even

✅ **Use when:** Short conditional assignments.

**9. Special Operators**

**a) Identity Operators**

Check if two variables point to the **same object in memory**.

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| is | Same object | a is b | True/False |
| is not | Not same object | a is not b | True/False |

a = [1,2]

b = [1,2]

print(a == b) # True (values same)

print(a is b) # False (different memory)

**b) Membership Operators**

Check if element exists in a collection.

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| in | True if element exists | 3 in [1,2,3] | True |
| not in | True if not exists | 4 not in [1,2,3] | True |

✅ **Use when:** Searching inside lists, strings, sets, dicts.

**🎯 Summary**

* **Maths:** Arithmetic, Assignment
* **Comparisons:** Relational, Equality
* **Logic:** Logical, Bitwise, Shift
* **Special:** Ternary, Identity, Membership

🔹 1. Arithmetic Operations

| Operator | Meaning | Example | Output |
| --- | --- | --- | --- |
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 5 - 3 | 2 |
| \* | Multiplication | 5 \* 3 | 15 |
| / | Division (float) | 5 / 2 | 2.5 |
| // | Floor Division | 5 // 2 | 2 |
| % | Modulus (remainder) | 5 % 2 | 1 |
| \*\* | Exponent (power) | 2 \*\* 3 | 8 |

🔹 Advanced use: pow(x, y, mod) → (x\*\*y) % mod (useful in cryptography).

🔹 2. Comparison Operations

| Operator | Meaning | Example | Output |
| --- | --- | --- | --- |
| == | Equal | 5 == 5 | True |
| != | Not equal | 5 != 3 | True |
| > | Greater than | 5 > 3 | True |
| < | Less than | 5 < 3 | False |
| >= | Greater or equal | 5 >= 5 | True |
| <= | Less or equal | 3 <= 5 | True |

🔹 Advanced use: Works with strings too → "apple" < "banana" ✅ (lexical order).

🔹 3. Logical Operations

| Operator | Meaning | Example | Output |
| --- | --- | --- | --- |
| and | True if both true | True and False | False |
| or | True if one true | True or False | True |
| not | Negation | not True | False |

🔹 Advanced use:

* Short-circuiting → x and y returns first false value / last value.

print(0 and 10) # 0

print(5 and 10) # 10

🔹 4. Bitwise Operations

(Works on binary representation)

| Operator | Meaning | Example | Binary Result |
| --- | --- | --- | --- |
| & | AND | 5 & 3 | 1 (101 & 011 = 001) |
| ` | ` | OR | `5 |
| ^ | XOR | 5 ^ 3 | 6 (101 ^ 011 = 110) |
| ~ | NOT (invert bits) | ~5 | -6 |
| << | Left shift | 5 << 1 | 10 |
| >> | Right shift | 5 >> 1 | 2 |

🔹 Advanced use: Efficient for cryptography, compression, image processing.

🔹 5. Assignment Operations

| Operator | Meaning | Example |
| --- | --- | --- |
| = | Assign value | x = 10 |
| += | Add and assign | x += 5 |
| -= | Subtract and assign | x -= 2 |
| \*= | Multiply and assign | x \*= 3 |
| /= | Divide and assign | x /= 2 |
| //= | Floor divide and assign | x //= 2 |
| %= | Modulus and assign | x %= 2 |
| \*\*= | Power and assign | x \*\*= 2 |
| `&=, | =, ^=, <<=, >>=` | Bitwise with assign |

🔹 6. Membership Operations

| Operator | Meaning | Example | Output |
| --- | --- | --- | --- |
| in | True if element exists | 3 in [1,2,3] | True |
| not in | True if not exists | 4 not in [1,2,3] | True |

🔹 7. Identity Operations

| Operator | Meaning | Example | Output |
| --- | --- | --- | --- |
| is | True if objects are same | x is y | Depends on memory address |
| is not | True if objects are not same | x is not y | Opposite |

🔹 Advanced use:

a = [1,2]

b = [1,2]

print(a == b) # True (values same)

print(a is b) # False (different memory)

🔹 8. Advanced Python Operators

* Walrus Operator (:=) → Assign inside expression

if (n := len("hello")) > 3:

print(n) # prints 5

* Ellipsis (...) → Placeholder in code, NumPy slicing

def todo(): ...

* Ternary Operator (a if cond else b) → One-line if-else

x = "Even" if 4 % 2 == 0 else "Odd"

Easy memory:

* Math → Arithmetic, Comparison, Assignment
* Logic → Logical, Bitwise
* Check → Membership, Identity

**🔢 Arithmetic Operators in Python with Different Data Types**

Arithmetic operators: + , - , \* , / , // , % , \*\*

**📌 1. Numbers (int, float, complex)**

👉 Normal mathematical operations.

a = 10

b = 4

print(a + b) # 14

print(a - b) # 6

print(a \* b) # 40

print(a / b) # 2.5 (float division)

print(a // b) # 2 (floor division)

print(a % b) # 2 (remainder)

print(a \*\* b) # 10000 (10 power 4)

✔️ **Used in:** All math-based programs.

**📌 2. Strings**

👉 Only **+** and **\*** work with strings.

| **Operator** | **Meaning** | **Example** | **Output** |
| --- | --- | --- | --- |
| + | Concatenation (join strings) | "Hello" + "World" | "HelloWorld" |
| \* | Repetition | "Hi" \* 3 | "HiHiHi" |

❌ Other operators (- , / , // , % , \*\*) don’t work with strings → Error.

✔️ **Used in:** Making sentences, repeating patterns.

**1. ASCII and Characters in Python**

* Every character in Python has a **numeric ASCII value**.
* Use **ord()** to get ASCII value of a character.
* Use **chr()** to get character from ASCII value.

# Get ASCII value

print(ord('A')) # 65

print(ord('a')) # 97

# Get character from ASCII

print(chr(65)) # 'A'

print(chr(97)) # 'a'

**2. String Comparison Using ASCII**

Python compares strings **lexicographically** using ASCII values:

s1 = "apple"

s2 = "banana"

# Compare first characters

print(ord(s1[0])) # 97

print(ord(s2[0])) # 98

print(s1 < s2) # True because 97 < 98

**Key points:**

* 'a' < 'A' → False (97 > 65 in ASCII)
* 'A' < 'B' → True
* 'abc' < 'abcd' → True (prefix matches, shorter string is smaller)

 ord(char) → ASCII of a character

 chr(number) → Character from A

**📌 3. Lists**

👉 Same as strings, only + and \* work.

❌ Others (- , / , % , \*\*) not supported.

✔️ **Used in:** Extending lists, repeating test data.

**📌 4. Tuples**

👉 Same behavior as lists.

print((1,2) + (3,4)) # (1, 2, 3, 4)

print((1,2) \* 3) # (1, 2, 1, 2, 1, 2)

✔️ **Used in:** Combining tuple data.

**📌 5. Booleans (True = 1, False = 0)**

👉 Treated as integers.

print(True + True) # 2

print(True \* 5) # 5

print(False + 10) # 10

✔️ **Used in:** Counters, logical math.

**📌 6. Sets & Dictionaries**

❌ + , - , \* , / not directly supported.  
👉 Instead, sets have their own operators like **union (|), intersection (&)**.  
Dictionaries cannot be added/multiplied with arithmetic operators.

**🎯 Summary Table**

| **Data Type** | **Supported Operators** | **Example** | **Result** |
| --- | --- | --- | --- |
| **int/float** | +, -, \*, /, //, %, \*\* | 10 % 3 | 1 |
| **string** | +, \* | "Hi" \* 2 | "HiHi" |
| **list** | +, \* | [1,2] + [3] | [1,2,3] |
| **tuple** | +, \* | (1,) \* 3 | (1,1,1) |
| **bool** | All (as 0/1) | True + False | 1 |
| **set** | No arithmetic (use ` | , &`) | `{1,2} |
| **dict** | Not supported | ❌ | Error |

In Python, you can **chain relational operators**:

10 < 20 < 30 # True

is same as:

(10 < 20) and (20 < 30)

**Examples:**

20 < 20 # False

20 < 20 < 30 # False

10 < 20 < 30 < 40 # True

10 < 20 < 30 > 40 > 50 < 20 # False

✅ Python keeps the middle terms only once → like math inequalities.

**1. What are Equality Operators?**

Equality operators are used to **compare values** for **equality or inequality**. They return a **boolean (True or False)**.

| **Operator** | **Meaning** |
| --- | --- |
| == | Equal to |
| != | Not equal to |

**2. Behavior with Different Data Types**

**a) Numbers (int, float, complex)**

* Compare **numerical values**.
* int and float are compatible (5 == 5.0 → True).
* complex supports only == and !=.

print(5 == 5.0) # True

print(2+3j == 2+3j) # True

print(2+3j != 3+2j) # True

**b) Strings**

* Compared **lexicographically**.

s1 = "apple"

s2 = "Apple"

print(s1 == s2) # False, case-sensitive

**c) Boolean**

* True = 1, False = 0. Can compare with numbers or other booleans.

x = True

print(x == 1) # True

print(x != False) # True

**d) Lists, Tuples**

* Compared **element-wise**.

[1,2,3] == [1,2,3] # True

(1,2) != (2,1) # True

**e) Sets**

* Compare **elements ignoring order**.

{1,2} == {2,1} # True

**3. Identity vs Equality**

* == → checks **value equality**
* is → checks **object identity** (memory location)

a = [1,2]

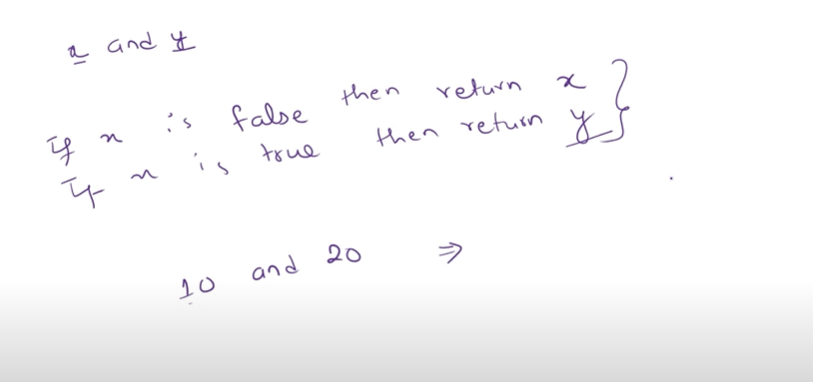
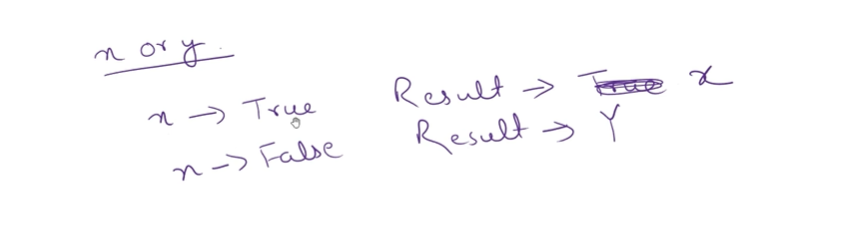
b = [1,2]

print(a == b) # True

print(a is b) # False

**✅ Summary**

* == → checks if values are equal
* != → checks if values are not equal
* Works with **all data types**, but complex types, sets, dicts, and None have special behavior.
* Use is when checking **object identity**, not equality.



**Logical Operators in Python – Advanced Details**

Python has three **logical operators**: and, or, and not. These operators are used to **combine boolean expressions** and can also work with **non-boolean objects** using Python’s concept of **truthy and falsy values**.

**1. Operators Overview**

| **Operator** | **Description** | **Returns** |
| --- | --- | --- |
| and | True if **both operands** are True | First falsy operand or last operand |
| or | True if **any operand** is True | First truthy operand or last operand |
| not | Inverts the truth value | Always returns Boolean (True or False) |

**2. Boolean Values**

Logical operators work **as expected** with True and False:

x = True

y = False

print(x and y) # False

print(x or y) # True

print(not x) # False

**3. Truthy and Falsy Values (Advanced Concept)**

Python treats **non-boolean objects** as True or False when used with logical operators:

**Falsy Values**

* False, None, 0, 0.0, 0j
* Empty sequences/collections: '', (), [], {}, set()

**Truthy Values**

* All other objects (non-empty strings, non-zero numbers, non-empty collections)

print(bool(0)) # False

print(bool(5)) # True

print(bool("")) # False

print(bool("Hi")) # True

**4. Behavior of and and or with Non-Boolean Values**

**and Operator**

* Returns **first falsy operand**, or **last operand** if all are truthy.

print(5 and 0) # 0 → first falsy

print(5 and 10) # 10 → both truthy, returns last

print("" and "Hi") # '' → first falsy

**or Operator**

* Returns **first truthy operand**, or **last operand** if all are falsy.

print(0 or 5) # 5 → first truthy

print("" or None) # None → all falsy

print(3 or 5) # 3 → first truthy

**not Operator**

* Always returns a **Boolean** value (True or False).

print(not 0) # True

print(not 5) # False

print(not "") # True

**5. Short-Circuit Evaluation (Advanced Feature)**

* Python **evaluates logical expressions left to right** and **stops as soon as the result is determined**.

def f():

print("Function called")

return True

# Short-circuit examples

print(False and f()) # f() NOT called

print(True or f()) # f() NOT called

**Benefits:**

* Improves **performance**
* Avoids unnecessary **function calls** or **side effects**

**6. Combining Logical Operators**

* You can combine and, or, and not in complex expressions.
* Use **parentheses** to control precedence.

x = 5

y = 10

z = 0

result = (x < y) and not (z > 0) or (y == 10)

print(result) # True

**Precedence Order:** not > and > or

**7. Logical Operators with Collections**

You can use and/or with **lists, tuples, dicts, sets**, using truthiness:

lst1 = [1,2,3]

lst2 = []

print(lst1 and lst2) # [] → lst2 is falsy

print(lst1 or lst2) # [1,2,3] → lst1 is truthy

* Useful for **default value selection**:

user\_input = ""

default = "Guest"

name = user\_input or default

print(name) # Guest

**8. Identity vs Logical Operators**

Logical operators check **truth value**, not **identity**.

* is checks object identity.
* and, or, not check **truthiness**.

a = None

b = 5

print(a or b) # 5 → b is truthy

print(a is None) # True

**9. Advanced Use Cases**

1. **Conditional expressions with default values**

username = input("Enter username: ") or "Anonymous"

1. **Chaining logical operations**

x = 5

y = 10

z = 15

if 0 < x < y and y < z:

print("Valid range")

1. **Avoiding function calls with short-circuit**

def expensive\_check():

print("Running check")

return True

if x > 10 or expensive\_check():

print("Passed")

# expensive\_check() not called if x > 10

**10. Summary Table (Advanced)**

| **Operator** | **Returns (boolean)** | **Behavior with Non-Boolean** | **Short-circuit** |
| --- | --- | --- | --- |
| and | First falsy or last truthy | Evaluates left to right, stops at first falsy | Yes |
| or | First truthy or last falsy | Evaluates left to right, stops at first truthy | Yes |
| not | Always Boolean | Converts operand to boolean | N/A |

**Bitwise Operators in Python – Advanced Details**

Bitwise operators **work at the binary level**, i.e., they manipulate the **individual bits** of integers.

Python supports these **bitwise operators**:

| **Operator** | **Symbol** | **Description** |
| --- | --- | --- |
| AND | & | Bitwise AND |
| OR | ` | ` |
| XOR | ^ | Bitwise XOR |
| NOT | ~ | Bitwise NOT (one’s complement) |
| LEFT SHIFT | << | Shift bits left (multiply by 2ⁿ) |
| RIGHT SHIFT | >> | Shift bits right (divide by 2ⁿ) |

**1. Bitwise AND &**

* Returns 1 only if **both corresponding bits** are 1.

a = 10 # 1010 in binary

b = 6 # 0110 in binary

print(a & b) # 2 → 0010

**2. Bitwise OR |**

* Returns 1 if **any corresponding bit** is 1.

a = 10 # 1010

b = 6 # 0110

print(a | b) # 14 → 1110

**3. Bitwise XOR ^**

* Returns 1 if **bits are different**.

a = 10 # 1010

b = 6 # 0110

print(a ^ b) # 12 → 1100

**4. Bitwise NOT ~**

* Returns **one’s complement**: flips all bits.
* Python integers are signed; formula: ~x = -x - 1

a = 10 # 1010

print(~a) # -11 → flips bits: 1010 → ...11110101 (-11 in two’s complement)

**5. Left Shift <<**

* Shifts bits to the left, fills **0s** on the right.
* Effectively multiplies by 2^n.

a = 5 # 0101

print(a << 1) # 10 → 1010 (01010)

print(a << 2) # 20 → 10100

**6. Right Shift >>**

* Shifts bits to the right, discards bits on the right.
* Effectively integer division by 2^n.

a = 20 # 10100

print(a >> 1) # 10 → 1010

print(a >> 2) # 5 → 0101

**7. Bitwise with Negative Numbers**

* Python uses **two’s complement** for negatives.
* Example:

a = -5

print(bin(a)) # -0b101

print(~a) # 4 → flips bits and adds sign

* Important: Left/right shifts preserve **sign** for signed integers.

**9. Summary Table**

| **Operator** | **Symbol** | **Effect** |
| --- | --- | --- |
| AND | & | Both bits 1 → 1, else 0 |
| OR | ` | ` |
| XOR | ^ | Bits different → 1, else 0 |
| NOT | ~ | Flip all bits (one’s complement) |
| Left Shift | << | Shift left, multiply by 2ⁿ |
| Right Shift | >> | Shift right, divide by 2ⁿ |

**1. Masking Bits (Picking certain bits only)**

👉 "Masking" means: keep some bits, hide others.  
We use & (AND) for this.

Example:

value = 0b1101 # 13

mask = 0b0110 # 6

print(value & mask)

Steps:

1101 (13)

& 0110 (6)

= 0100 (4)

👉 Only common 1s are kept. Result = 4

**2. Setting Bits (Turn ON a bit)**

👉 Use | (OR) to make a certain bit 1.

Example:

value = 0b1000 # 8

value |= 0b0010 # set 2nd bit

print(bin(value))

Steps:

1000 (8)

| 0010 (2)

= 1010 (10)

👉 Bit is set (turned ON).

**3. Toggling Bits (Flip a bit ON ↔ OFF)**

👉 Use ^ (XOR) to **flip** a bit.

Example:

value = 0b1010 # 10

value ^= 0b1111 # toggle all bits

print(bin(value))

Steps:

1010 (10)

^ 1111 (15)

= 0101 (5)

👉 All bits flipped.

**4. Multiplying / Dividing by 2**

👉 Use << (left shift) for multiply, >> (right shift) for divide.

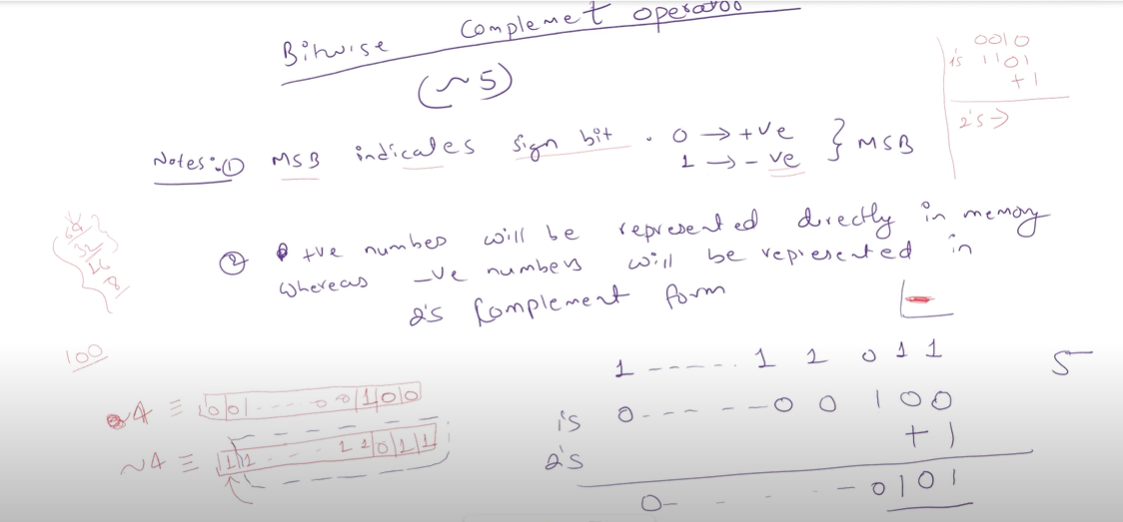
Example:

x = 7

print(x << 1) # 14 → 7\*2

print(x << 2) # 28 → 7\*4

print(x >> 1) # 3 → 7//2



Operator Comparison: Python vs Other Languages

1. Arithmetic Operators

| Operation | Python | C / C++ / Java / JS | Notes |
| --- | --- | --- | --- |
| Addition | + | + | Same |
| Subtraction | - | - | Same |
| Multiplication | \* | \* | Same |
| Division | / | / | Python → float always, C/Java → int if both operands are int |
| Floor Division | // | No direct operator (use / for int division in C/Java < Java 8) | Special in Python |
| Modulus | % | % | Same |
| Exponent | \*\* | pow(a,b) (C/Java), Math.pow() (JS/Java) | Only Python has \*\* |

2. Relational / Comparison Operators

| Operation | Python | C / C++ / Java / JS | Notes |
| --- | --- | --- | --- |
| Equal | == | == | Same meaning |
| Not Equal | != | != (C/Java), !== in JS | Python only has != |
| Greater | > | > | Same |
| Less | < | < | Same |
| Greater or Equal | >= | >= | Same |
| Less or Equal | <= | <= | Same |

3. Logical Operators

| Operation | Python | C / C++ / Java / JS | Notes |
| --- | --- | --- | --- |
| AND | and | && | Different syntax |
| OR | or | ` |  |
| NOT | not | ! | Different syntax |
| Bitwise AND | & | & | Same |
| Bitwise OR | ` | ` | ` |
| Bitwise XOR | ^ | ^ | Same |
| Bitwise NOT | ~ | ~ | Same |

4. Assignment Operators

| Operation | Python | C / C++ / Java / JS | Notes |
| --- | --- | --- | --- |
| Assign | = | = | Same |
| Add & Assign | += | += | Same |
| Subtract & Assign | -= | -= | Same |
| Multiply & Assign | \*= | \*= | Same |
| Divide & Assign | /= | /= | Same but Python keeps float |
| Floor Divide & Assign | //= | Not available | Python-only |
| Modulus & Assign | %= | %= | Same |
| Power & Assign | \*\*= | No direct (use pow) | Python-only |
| Bitwise AND & Assign | &= | &= | Same |
| Bitwise OR & Assign | ` | =` | ` |
| Bitwise XOR & Assign | ^= | ^= | Same |
| Left Shift & Assign | <<= | <<= | Same |
| Right Shift & Assign | >>= | >>= | Same |

5. Identity & Membership

| Operation | Python | C / C++ / Java / JS | Notes |
| --- | --- | --- | --- |
| Identity (is / is not) | is, is not | No direct equivalent | Checks object identity |
| Membership | in, not in | No direct equivalent | Checks in list, string, set, dict |

6. Special

| Feature | Python | C / C++ / Java / JS |
| --- | --- | --- |
| Chained Comparisons | a < b < c | Not available (must use a < b && b < c) |
| Truthy/Falsy | Any object can be truthy/falsy | In C/Java, only numbers or boolean |
| Operator Overloading | Supported (with magic methods like \_\_add\_\_) | Supported in C++ but not in Java |

✅ Key Differences (Easy Memory)

* Python uses words → and, or, not instead of &&, ||, !.
* Python has extra operators → // (floor divide), \*\* (power).
* Python comparisons can chain → a < b < c.
* Python has is and in, which other languages don’t.
* Division / in Python → always float. In C/Java, int division gives int.

**🌟 Conditional / Ternary Operator in Python (Advanced Notes)**

**1. Syntax**

value\_if\_true if condition else value\_if\_false

👉 It evaluates the **condition** first:

* If True → returns value\_if\_true.
* If False → returns value\_if\_false.

**2. Basic Example**

x = 7

result = "Even" if x % 2 == 0 else "Odd"

print(result) # Odd

**3. Nested Conditional (Multiple Choices)**

You can chain multiple ternary operators:

x = -5

result = "Positive" if x > 0 else "Zero" if x == 0 else "Negative"

print(result) # Negative

👉 Equivalent to:

if x > 0:

result = "Positive"

elif x == 0:

result = "Zero"

else:

result = "Negative"

**4. Inside Expressions**

Ternary operators can be used **inside lists, dicts, function calls** etc.

**(a) In a list**

nums = [1, 2, 3, 4, 5]

parity = ["Even" if n % 2 == 0 else "Odd" for n in nums]

print(parity) # ['Odd', 'Even', 'Odd', 'Even', 'Odd']

**(b) In dictionary values**

x = 10

info = {"status": "Positive" if x > 0 else "Non-Positive"}

print(info) # {'status': 'Positive'}

**(c) Inside a function**

def check\_age(age):

return "Adult" if age >= 18 else "Minor"

print(check\_age(20)) # Adult

print(check\_age(12)) # Minor

**5. With Functions (Lazy Evaluation)**

Only the required branch is executed.

def heavy\_true():

print("True branch executed")

return "YES"

def heavy\_false():

print("False branch executed")

return "NO"

x = 5

result = heavy\_true() if x > 0 else heavy\_false()

print(result)

# Output:

# True branch executed

# YES

👉 The False branch was **never executed**.

**6. Multiple Assignments**

You can assign different values depending on condition in a single line.

x, y = (1, "Positive") if 5 > 0 else (0, "Non-Positive")

print(x, y) # 1 Positive

**7. With Lambda Functions**

Ternary operator fits inside **lambda** functions (anonymous functions).

check = lambda x: "Even" if x % 2 == 0 else "Odd"

print(check(7)) # Odd

print(check(12)) # Even

**8. With Loops**

Use it in loops for compact decisions.

for i in range(5):

print("Even" if i % 2 == 0 else "Odd", end=" ")

# Output: Even Odd Even Odd Even

**9. With Complex Conditions**

You can mix conditions inside ternary operator.

x = 15

result = "FizzBuzz" if x % 3 == 0 and x % 5 == 0 else "Fizz" if x % 3 == 0 else "Buzz" if x % 5 == 0 else str(x)

print(result) # FizzBuzz

**✅ Key Points to Remember**

* Python ternary: A if condition else B.
* **Only one branch is executed** (short-circuit).
* Can be used in **assignments, functions, lists, dicts, loops**.
* Supports **nested conditions**.
* Great for **compact code**, but don’t overuse (can reduce readability).

**🌟 Special Operators in Python**

Python has **2 main special operators**:

**1. Identity Operators**

👉 Used to compare **memory address (object identity)**, not values.

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| is | True if both refer to the **same object** | a = [1,2]; b = a; a is b | True |
| is not | True if they are **different objects** | a = [1,2]; b = [1,2]; a is not b | True |

🔎 Example:

x = [1,2,3]

y = [1,2,3]

print(x == y) # True (values equal)

print(x is y) # False (different memory)

**2. Membership Operators**

👉 Used to check **if a value exists in a sequence** (list, tuple, string, dict, set).

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| in | True if element is present | "a" in "ram" | True |
| not in | True if element is absent | 10 not in [1,2,3] | True |

🔎 Example:

name = "Ram Prasad"

print("Ram" in name) # True

print("ram" not in name) # True (case-sensitive)

**✅ Quick Summary Table**

| **Category** | **Operators** | **Use** |
| --- | --- | --- |
| Identity | is, is not | Compare **object identity (memory)** |
| Membership | in, not in | Check if **element exists** in a sequence |

👉 These are called **special operators** because they don’t directly do math or logic, but handle **objects and sequences**.

**🌟 Operator Procedure in Python**

**1. Operator Precedence (Which runs first?**

Some operators have higher priority than others.  
👉 Example:

x = 10 + 5 \* 2

print(x) # 20 (multiplication runs before addition)

**Order (high → low):**

1. () → Parentheses (highest)
2. \*\* → Exponent
3. +x, -x, ~x → Unary plus/minus, bitwise NOT
4. \*, /, //, % → Multiplication, division, floor, modulo
5. +, - → Addition, subtraction
6. <<, >> → Bitwise shifts
7. & → Bitwise AND
8. ^ → Bitwise XOR
9. | → Bitwise OR
10. <, >, <=, >=, ==, != → Comparisons
11. is, is not, in, not in → Identity, Membership
12. not → Logical NOT
13. and → Logical AND
14. or → Logical OR
15. =, +=, -=, \*=, ... → Assignment (lowest)

**BODMAS Rule in Python**

**BODMAS = Brackets → Orders → Division/Multiplication → Addition → Subtraction**

**2. Associativity (If same precedence, which side first?)**

* **Left-to-right** (most operators)
* **Right-to-left** (exponent \*\*, assignment =, ternary if-else)

👉 Example:

print(2 \*\* 3 \*\* 2) # 2 \*\* (3 \*\* 2) = 2 \*\* 9 = 512

**3. Operator Evaluation Procedure**

1. **Check precedence** (which operator is higher).
2. **Check associativity** (direction for same precedence).
3. **Evaluate operands** step by step.
4. **Return final result**.

👉 Example:

x = 2 + 3 \* 4 \*\* 2

# Step 1: Exponent → 4 \*\* 2 = 16

# Step 2: Multiplication → 3 \* 16 = 48

# Step 3: Addition → 2 + 48 = 50

print(x) # 50

**4. Operator Overloading (Advanced)**

In Python, operators are implemented using **special methods** (dunder methods).  
You can define your own behavior for custom classes.

👉 Example:

class Box:

def \_\_init\_\_(self, value):

self.value = value

def \_\_add\_\_(self, other): # Overload +

return Box(self.value + other.value)

a = Box(10)

b = Box(20)

c = a + b

print(c.value) # 30

**✅ Summary Table**

| **Step** | **Concept** | **Meaning** |
| --- | --- | --- |
| 1 | **Precedence** | Which operator runs first |
| 2 | **Associativity** | Direction (left → right or right → left) |
| 3 | **Evaluation** | Solve step by step |
| 4 | **Overloading** | Customize operators for objects |

**Day7:**

**Input in Python**

🔹 1. Basic Input

* Python uses input() to take input from the user.
* Always returns a string type.

name = input("Enter your name: ")

print("Hello", name)

👉 If you enter Ram, output will be:

Hello Ram

🔹 2. Type Conversion

Since input() gives string, convert it when needed.

age = int(input("Enter age: ")) # integer

height = float(input("Enter height: ")) # float

👉 If you enter 21 and 5.6, you get 21 (int), 5.6 (float).

🔹 3. Multiple Inputs (One Line)

a, b = input("Enter two values: ").split()

print("a =", a, "b =", b)

👉 If you enter 10 20 → a=10, b=20

✔ With type conversion:

x, y = map(int, input("Enter two numbers: ").split())

print("Sum =", x + y)

🔹 4. List Input

Take many numbers at once into a list.

nums = list(map(int, input("Enter numbers: ").split()))

print(nums)

👉 Input: 1 2 3 4 5 → Output: [1, 2, 3, 4, 5]

🔹 5. Advanced Input

(a) Using split(",")

names = input("Enter names: ").split(",")

print(names)

👉 Input: Ram,Sita,Hari → ['Ram', 'Sita', 'Hari']

3 4

→ Output: [[1, 2], [3, 4]]

✅ Summary Table

| Method | Purpose | Example Input | Example Output |
| --- | --- | --- | --- |
| input() | Single input | Ram | "Ram" |
| int(input()) | Numeric input | 21 | 21 |
| .split() | Multiple values | 10 20 | ["10", "20"] |
| map(int, input().split()) | Multiple integers | 10 20 | 10, 20 |
| list(map(int, ...)) | List input | 1 2 3 | [1, 2, 3] |
| .split(",") | Comma-separated | Ram,Sita | ["Ram","Sita"] |
| List comprehension | Processed input | 2 3 | [4, 9] |
| sys.stdin.read() | Fast input | (big data) | List of strings |

**Day8:**

**Indentation in Python**

Indentation means spaces or tabs at the beginning of a line to define code blocks.  
In Python, indentation is mandatory (unlike C, C++, or Java where { } are used).

Example (Correct):

if True:

print("Yes")

print("Inside if block")

print("Outside if block")

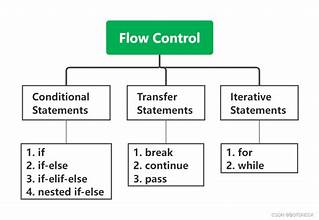
Example (Error):

if True:

print("Yes") # ❌ IndentationError

🔹 Advantages of Indentation in Python

1. Improves Readability  
   Code looks structured and is easier to understand.
2. for i in range(3):
3. if i % 2 == 0:
4. print(i, "is even")
5. No Extra Braces Needed  
   Other languages require { }, but Python relies only on indentation, making syntax cleaner.
6. Consistency  
   Python enforces consistent style (PEP8 recommends 4 spaces), so codebases remain uniform.
7. Fewer Errors  
   Since indentation is required, you can’t forget to close braces or accidentally misplace code blocks.
8. Clean and Professional Syntax  
   Python is famous for being readable and “English-like,” making it great for collaboration and teaching.



**📘 Flow Control / Control Flow / Control Structures in Python**

Flow control = defines **how the program executes statements** instead of just top-to-bottom.

**🔹 1. Conditional Statements (Decision Making)**

Used to execute code **based on conditions**.

* if → single condition
* if-else → two-way decision
* if-elif-else → multiple decisions
* Nested if → condition inside condition

✅ Example:

x = 10

if x > 0:

print("Positive")

else:

print("Negative")

**🔹 2. Looping Statements (Iteration)**

Used for **repetition** of tasks.

* for → iterate over sequence (list, string, range)
* while → repeat until condition false
* Nested loops → loop inside loop

✅ Example:

for i in range(3):

print(i)

**🔹 3. Jump/Transfer Statements**

Alter the normal flow inside loops.

* break → exit loop completely
* continue → skip current iteration
* pass → placeholder (does nothing)

**1. Conditional Statements (Decision Making)**

Conditional statements are used to **make decisions in a program**. They allow code to execute **only when certain conditions are true**.  
👉 Python uses **if, if-else, if-elif-else, and nested if** for decision making.

**✅ Basic Rules**

* Condition is written inside **parentheses ()** after if (not compulsory but good practice).
* Condition must return a **Boolean value** (True or False).
* Code block under condition must be indented.

**1. if Statement**

Executes a block **only if condition is True**.

**Syntax:**

if condition:

statement(s)

**Example:**

age = 18

if age >= 18:

print("You are eligible to vote")

**2. if-else Statement**

Provides **two-way decision**.

* If condition is true → run if block.
* Else → run else block.

**Syntax:**

if condition:

statement(s)

else:

statement(s)

**Example:**

num = -5

if num >= 0:

print("Positive number")

else:

print("Negative number")

**3. if-elif-else Statement**

Used for **multiple conditions** (multi-way branching).

* First condition checked, if true → stop checking.
* Else, next elif checked.
* If none true → else executed.

**Syntax:**

if condition1:

statement(s)

elif condition2:

statement(s)

elif condition3:

statement(s)

else:

statement(s)

**Example:**

marks = 72

if marks >= 90:

print("Grade A")

elif marks >= 75:

print("Grade B")

elif marks >= 50:

print("Grade C")

else:

print("Fail")

**4. Nested if Statement**

if statement **inside another if**.  
Used when multiple related conditions must be checked.

**Syntax:**

if condition1:

if condition2:

statement(s)

**Example:**

x = 15

if x > 10:

if x % 2 == 0:

print("Greater than 10 and Even")

else:

print("Greater than 10 and Odd")

**5. Short-hand if (Ternary Operator Style)**

When only one statement is inside if or if-else, it can be written in **one line**.

**Example:**

x = 5

if x > 0: print("Positive") # short-hand if

# short-hand if-else

y = 10

print("Even") if y % 2 == 0 else print("Odd")

**day9:**

1. **Looping Statements (Iteration) in Python**

**🔹 1. Introduction**

* **Looping (Iteration)** means executing a block of code **repeatedly** until a condition is satisfied.
* It is used to reduce code repetition and make programs efficient.
* In Python, there are **two main types of loops**:
  1. **for loop**
  2. **while loop**

**🔹 2. for Loop**

**👉 Definition:**

* The **for loop** is used when we know the **number of times** to repeat.
* It works directly with sequences (list, tuple, string, dictionary, set, range).

**👉 Syntax:**

for variable in sequence:

statement(s)

**for** → keyword to start the loop.

**variable** → takes one element from the sequence at a time.

**in** → keyword meaning “from”.

**sequence** → list, tuple, string, range, dictionary, etc.

A **sequence** can be:

* **list** → [10, 20, 30]
* **tuple** → (1, 2, 3)
* **string** → "hello"
* **dictionary** → {key:value} (iterates over keys)
* **set** → {1, 2, 3}
* **range()** → range(5)

**statement(s)** → code to execute for each element.

**👉 Example 2: Using range()**

for i in range(1, 6): # runs from 1 to 5

print(i)

**Output:** 1 2 3 4 5

**🔹 3. while Loop**

**👉 Definition:**

* The **while loop** is used when we **don’t know in advance** how many times to repeat.
* It runs until the given condition becomes **False**.

**👉 Syntax:**

while condition:

statement(s)

**👉 Example:**

count = 1

while count <= 5:

print("Count:", count)

count += 1

**Output:**

Count: 1

Count: 2

Count: 3

Count: 4

Count: 5

**4. Loop Control Statements**

Sometimes we need to **control the flow** inside loops.

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

for i in range(1, 10):

if i == 5:

break

print(i)

**Output:** 1 2 3 4

1. **continue** → Skip the current iteration and go to the next one.

for i in range(1, 6):

if i == 3:

continue

print(i)

**Output:** 1 2 4 5

1. **pass** → Does nothing (used as placeholder).

for i in range(3):

pass # future code can be written here

**🔹 5. Nested Loops**

A loop inside another loop.

for i in range(2): # outer loop

for j in range(3): # inner loop

print(f"i={i}, j={j}")

**Output:**

i=0, j=0

i=0, j=1

i=0, j=2

i=1, j=0

i=1, j=1

i=1, j=2

**6 What is for...else?**

In Python, a for loop can have an else block.

👉 The else block runs only if the loop finishes completely (without break).  
👉 If the loop is stopped by break, then the else block is skipped.

🔹 Example 1: Normal loop (no break)

for i in range(5):

print(i)

else:

print("Loop finished!")

✅ Output:

0

1

2

3

4

Loop finished!

👉 Here, the loop ended normally (no break used), so the else part executed.

🔹 Example 2: Loop with break

for i in range(5):

if i == 3:

break

print(i)

else:

print("Loop finished!")

✅ Output:

0

1

2

👉 Here, the loop stopped at i == 3 because of break.  
👉 That’s why the else block did NOT run.

🔹 Example 3: Searching an item (Practical Use)

numbers = [2, 4, 6, 8, 10]

for n in numbers:

if n == 7:

print("Found!")

break

else:

print("Not Found!")

✅ Output:

Not Found!

👉 Explanation:

* Loop checked all numbers.
* Did not find 7.
* Since no break happened, else executed → "Not Found!"

**(b) enumerate() → get index with value**

fruits = ["apple", "banana", "mango"]

for index, fruit in enumerate(fruits):

print(index, fruit)

**Output:**

0 apple

1 banana

2 mango

**(c) zip() → loop through multiple sequences at once**

names = ["Ram", "Sita", "Hari"]

marks = [90, 85, 88]

for n, m in zip(names, marks):

print(n, m)

**Output:**

Ram 90

Sita 85

Hari 88

**(d) Looping through Dictionary**

student = {"name": "Sita", "age": 20, "marks": 85}

for key, value in student.items():

print(key, ":", value)

**Output:**

name : Sita

age : 20

marks : 85

**(e) List Comprehension (One-line Loop)**

* Compact way to write loops and create lists.

squares = [x\*x for x in range(1, 6)]

print(squares)

**Output:** [1, 4, 9, 16, 25]

**🔹 7. Flowchart of Looping**

**For loop flow:**

Start → sequence available? → pick next item → execute body → repeat → end

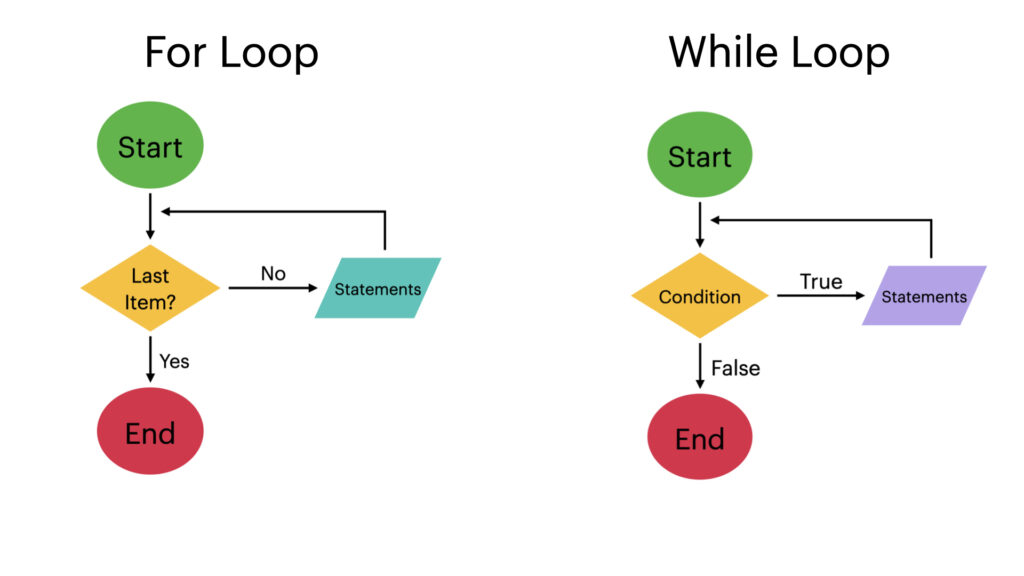
**While loop flow:**

Start → check condition → True? → execute body → update → repeat

↓

False → end

* Advanced Python provides **for-else, enumerate, zip, dictionary iteration, and list comprehensions** for cleaner looping.



**3. Loop Control Statements in Python**

Sometimes we need to **change the normal flow** of a loop.  
Python provides **3 control statements**:

1. **break**
2. **continue**
3. **pass**

**🔹 1. break Statement**

**Definition:**

* The break statement is used to **exit the loop immediately**, even if the loop condition is still true or the sequence is not finished.
* After break, control goes **outside the loop**.

**Syntax:**

for/while loop:

if condition:

break

statement(s)

**Example:**

for i in range(1, 10):

if i == 5:

break

print(i)

✅ **Output:**

1

2

3

4

**Explanation:**

* Loop runs from 1 to 9.
* When i == 5, the break statement stops the loop immediately.
* So numbers 5 to 9 are not printed.

**🔹 2. continue Statement**

**Definition:**

* The continue statement is used to **skip the current iteration** of the loop.
* The loop does not stop; it **continues with the next iteration**.

**Syntax:**

if condition:

continue

statement(s)

for/while loop:

**Example:**

for i in range(1, 6):

if i == 3:

continue

print(i)

✅ **Output:**

1

2

4

5

**Explanation:**

* Loop runs from 1 to 5.
* When i == 3, the continue statement skips that iteration.
* So 3 is not printed.

**🔹 3. pass Statement**

**Definition:**

* The pass statement **does nothing**.
* It is used as a **placeholder** when we don’t want to write code yet, but syntactically a statement is required.

**Syntax:**

for/while loop:

pass

**Example:**

for i in range(3):

pass # future code can be written here

✅ **Output:**

(No output)

**Explanation:**

* The loop runs three times, but pass tells Python to "do nothing" each time.
* It is useful when writing code structure but not filling logic yet.

**🔹 Flow Control Diagrams**

**For break:**

Start → Loop condition → True → Check break condition

↓Yes → Exit loop immediately

↓No → Execute body → Repeat

**For continue:**

Start → Loop condition → True → Check continue condition

↓Yes → Skip this iteration → Next loop

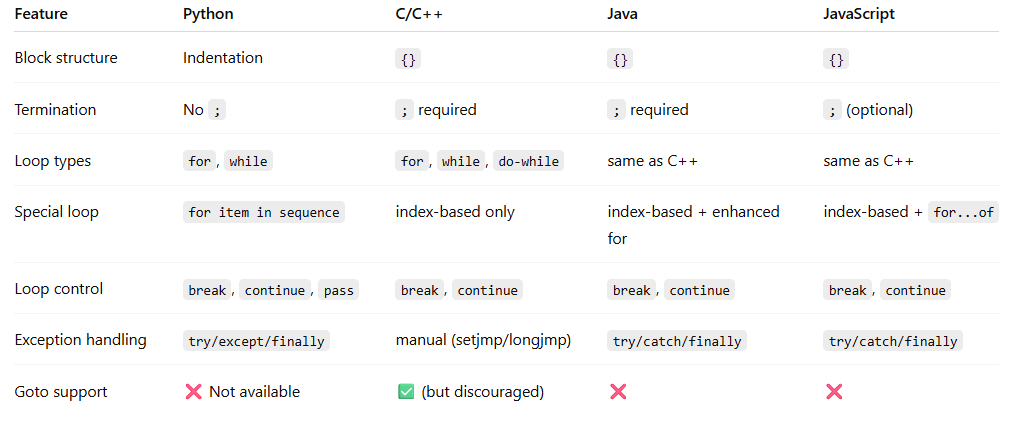
↓No → Execute body → Repeat

**For pass:**

Start → Loop condition → True → Do nothing → Next iteration

**🔹 Key Differences**

| **Statement** | **Action** |
| --- | --- |
| **break** | Terminates the loop immediately. |
| **continue** | Skips the current iteration and moves to the next one. |
| **pass** | Does nothing (placeholder). |



#practice question

prime number

n=int(input("enter a number:"))

if n<2:

   print("not prime number")

else:

    for i in range(2,n):

        if n%i==0:

           print("not prime number:")

           break

    else:

      print("its prime number")

 📌 Example 1:

Number = 145

Digits → 1, 4, 5

1!=1

4!=24

5!=120

1+24+120=145

✅ Yes, 145 is a Strong Number

n=int(input("enter a number:"))

temp=n

sum=0

while temp>0:

    d=temp%10

    f=1

    for i in range(1, d+1):

         f=f\*i

    sum=sum +f

    temp= temp//10  #It is used to remove the last digit of a number after processing it.

if sum==n:

     print("it's a strong number")

else:

     print("not storng ")

**palindrome**

n=int(input("enter number:"))

temp=n

rev=0

while temp>0:

     digit=temp%10

     rev=rev\*10+digit

     temp=temp//10

if rev==n:

     print(n,"is palindrome number")

else:

     print("not palindrome"

#armstrong

n=int(input("enter number:"))

p=len(str(n))

temp=n

sum=0

while temp>0:

    digit=temp%10

    s=digit\*\*p

    sum=sum+s

    temp//=10

if n==sum:

    print(n,"is a armstrong ")

else:

    print("not a armstrong")

**🔹 What is a Harshad (Niven) Number?**

A **Harshad number** is a number that is **divisible by the sum of its digits**.

**Formula:**



**🔹 Examples**

1. **18**

* Digits: 1 + 8 = 9
* 18 ÷ 9 = 2 → divisible ✅
* So, **18 is a Harshad number**

# Harshad number check using digits

n = int(input("Enter a number: "))

temp = n # Temporary variable to extract digits

sum\_digits = 0 # To store sum of digits

# Step 1: Sum all digits

while temp > 0:

digit = temp % 10 # Get last digit

sum\_digits += digit # Add digit to sum

temp //= 10 # Remove last digit

# Step 2: Check divisibility

if n % sum\_digits == 0:

print(n, "is a Harshad (Niven) Number")

else:

print(n, "is Not a Harshad Number")

**PERFECT NUMBER**

**🔹 What is a Perfect Number?**

A **perfect number** is a number that is **equal to the sum of its proper divisors** (all positive divisors excluding the number itself).

* Example: 6 → Divisors: 1, 2, 3 → Sum = 1 + 2 + 3 = 6 ✅
* Example: 28 → Divisors: 1, 2, 4, 7, 14 → Sum = 28 ✅

**🔹 Python Code**

# Perfect number check

n = int(input("Enter a number: "))

sum\_div = 0

# Step 1: Find sum of proper divisors

for i in range(1, n): # Loop through numbers less than n

if n % i == 0: # If i divides n completely

sum\_div += i # Add i to sum

# Step 2: Compare sum with original number

if sum\_div == n:

print(n, "is a Perfect number")

else:

print(n, "is Not a Perfect number")

**🔹 How It Works (Example: 28)**

1. n = 28, sum\_div = 0
2. Loop i from 1 to 27:
   * 1 → 28 % 1 = 0 → sum\_div = 0 + 1 = 1
   * 2 → 28 % 2 = 0 → sum\_div = 1 + 2 = 3
   * 3 → 28 % 3 ≠ 0 → ignore
   * 4 → 28 % 4 = 0 → sum\_div = 3 + 4 = 7
   * 7 → 28 % 7 = 0 → sum\_div = 7 + 7 = 14
   * 14 → 28 % 14 = 0 → sum\_div = 14 + 14 = 28
3. Compare: 28 == 28 → Perfect number ✅

**🔹 Example Runs**

Input: 6

Output: 6 is a Perfect number

Input: 28

Output: 28 is a Perfect number

Input: 12

Output: 12 is Not a Perfect number

**------------------- TWIN PRIME NUMBERS -------------------**

**🔹 What are Twin Primes?**

**Twin primes** are **pairs of prime numbers that differ by 2**.

* Examples: (3, 5), (5, 7), (11, 13), (17, 19)

**Rules:**

1. Both numbers must be **prime**.
2. Difference between them must be exactly 2.

**🔹 Python Code**

# Twin prime numbers check

a = int(input("Enter first number: "))

b = int(input("Enter second number: "))

# Function to check if a number is prime

def is\_prime(x):

if x < 2:

return False

for i in range(2, x):

if x % i == 0:

return False

return True

# Check if both numbers are prime and difference is 2

if is\_prime(a) and is\_prime(b) and abs(a - b) == 2:

print(a, "and", b, "are Twin Primes")

else:

print(a, "and", b, "are Not Twin Primes")

**🔹 How It Works (Example: 11 and 13)**

1. Check if 11 is prime → Yes
2. Check if 13 is prime → Yes
3. Difference: 13 - 11 = 2 → satisfies twin prime condition ✅

**🔹 Example Runs**

Input: a = 11, b = 13

Output: 11 and 13 are Twin Primes

Input: a = 17, b = 19

Output: 17 and 19 are Twin Primes

Input: a = 10, b = 12

Output: 10 and 12 are Not Twin Primes

**🔹 Key Points**

* **Perfect number** → sum of divisors = number
* **Twin primes** → both numbers prime & difference = 2
* % → checks divisibility
* abs(a-b) → finds difference

**Day10:**

String:

**1. Basic String Initialization:**

The explanation is accurate:

* Strings can be created using **single quotes** ('), **double quotes** ("), or **triple quotes** (''' or """).
* Python doesn't distinguish between single and double quotes—they are both used for creating strings.

a = 'a' # Single quote

g = "yuresh" # Double quote

Both variables are correctly identified as strings, and the type() function confirms this.

**2. Multiline Strings:**

The explanation is correct:

* Triple quotes (''' or """) are used to create multiline strings.
* Example with both triple single and triple double quotes is included.

a = '''my name is

yuresh gurung'''

print(a)

a = """my name is

yuresh gurung"""

print(a)

Both are valid, and the output correctly displays the string over multiple lines.

**3. Escape Sequences:**

The use of escape sequences like \n, \", and \' is explained properly. The examples are accurate:

* \n for newline.
* \" for escaped double quotes.
* \' for escaped single quotes.

s = "my name is \n yuresh gurung"

print(s)

s = "my name is \"yuresh gurung\""

print(s)

s = "it\'s a dog"

print(s)

1. **Accessing Characters and String Indexing:**

This section is correct:

* String indexing starts at 0 (positive index) and -1 (negative index).
* The code loops through the string and displays the index of each character, both positive and negative.

S = “my name is yuresh”

I = 0

print(“Index of given string in both positive and negative index:”)

for x in s:

print(f”Character {x} at positive index {i} and negative index {I – len(s)}”)

I += 1

The explanation about indexing and negative indexing is well covered.

1. **Slicing Strings:**

This part is also accurate:

* The slicing operator [start:end:step] works as expected.
* Reversing a string with [::-1] and using steps for slicing ([::-2]) are correctly explained.

S = “yuresh is a good person”

print(s[0:5:2]) # Slicing with a step of 2

print(s[::-1]) # Reverse the string

print(s[::-2]) # Step of 2: every other character in reverse

You also mention reverse iteration using reversed(s) and explain that the slice s[7:2:-1] works correctly with a negative step.

1. **Mathematical Operations with Strings:**

The explanation about string concatenation (+) and repetition (\*) is correct:

print(“hi” + “ram”) # Concatenate

print(“yuresh” \* 4) # Repeat string 4 times

print(len(“yuresh”)) # Length of string

The len() function and . \_\_len\_\_() method are mentioned correctly, though len() is more commonly used for checking string length.

1. **Membership Operators:**

The explanation of using the in and not in operators to check for substrings is correct:

a = “yuresh”

print(“y” in a) # True

print(“y” not in a) # False

This is a standard use of the membership operators.

**8. String Comparison:**

The use of comparison operators (==, <, >, etc.) is explained well. You also demonstrated case-sensitive comparisons:

a = "Ram"

b = "ram"

print(a == b) # False

print(a < b) # True (lexicographically)

The example compares two strings lexicographically, as expected.

**9. String Stripping:**

The strip(), lstrip(), and rstrip() methods are described accurately for removing whitespace from strings. The example of checking user input with strip() is correct:

a = input("Enter any name: ")

if a.strip() == "ram":

print("Name is correct")

else:

print("Wrong name")

This checks for leading/trailing spaces before comparing.

**10. Finding Substrings:**

The section correctly explains how to use .find() and .rfind() to find substrings, and .index() and .rindex() to get the index of substrings. The find() method returns -1 when the substring is not found, while .index() raises an exception.

a = "yuresh"

print(a.find("u")) # 0

print(a.rfind("u")) # 0

# Index method raises an error if not found

try:

print(a.index("t"))

except ValueError:

print("Substring not found.")

This section is well-covered and correctly explains error handling with .index().

**11. Counting and Replacing Substrings:**

The .count() and .replace() methods are explained properly.

a = "ram"

print(a.count("r")) # 1

a = "ram"

print(a.replace("r", "k")) # "kam"

Both methods work as expected, and the explanation is clear.

**12. Splitting and Joining Strings:**

You correctly explained the .split() and .join() methods, and the examples are correct:

a = " my name is ram gurung"

split\_a = a.split() # Splits into list of words

print(split\_a)

w = ['my', 'name', 'is', 'ram', 'gurung']

joined\_a = ' '.join(w) # Join list into a string

print(joined\_a)

This section covers splitting a string into a list and joining a list back into a string effectively.

**13. Case Changes:**

You correctly explained the different case transformation methods:

s = "my name is yures gurung"

print(s.upper()) # "MY NAME IS YURES GURUNG"

print(s.lower()) # "my name is yures gurung"

print(s.swapcase()) # "MY NAME IS YURES GURUNG"

print(s.title()) # "My Name Is Yures Gurung"

print(s.capitalize()) # "My name is yures gurung"

All methods work as expected, and you've described them properly.

**14. String Property Checks:**

The isalnum(), isalpha(), isdigit(), etc., methods are explained clearly:

print("ram123".isalnum()) # True

print("ram123".isalpha()) # False

print("ram123".isdigit()) # False

print("ram123".islower()) # True

The usage and output of these methods are correctly explained.

**15. String Formatting:**

You have provided multiple ways to format strings, including f-strings, .format(), and older formatting methods.

name = "yuresh"

age = 21

print(f"My friend name is {name} and age is {age}")

print("My friend name is {0} and age is {1}".format(name, age))

print("My friend name is {a} and age is {b}".format(a=name, b=age))

print("My friend name is {:105d}".format(123)) # Right-align with width 105

The explanation and examples are correct, and all formatting methods work as described.

**16. Viewing Available String Methods:**

The dir(str) method correctly lists all available string methods:

print(dir(str)) # Shows all methods available for string objects

This is an appropriate way to view all string-related methods.

**Day 11**

**Python List – Full Advanced Notes**

**🔹 1. Definition of List**

* A **List** is a **group of elements treated as a single entity**.
* It is one of the most widely used **sequence data structures** in Python.

👉 **Key Points**:

* **Duplication allowed** (same element can appear multiple times).
* **Insertion order preserved** ( order doesn’t change automatically).
* **Heterogeneous objects allowed** (A single list can store elements of **different data types** (int, float, string, boolean, even another list, etc.).
* **Dynamic** (can grow or shrink).

list1 = [10, 23, 51, 45, 10, "radheradhe"]

print(list1) # [10, 23, 51, 45, 10, 'radheradhe']

print(type(list1)) # <class 'list'>

**🔹 2. Properties of List**

* Syntax: **[]** (square brackets).
* Ordered (preserves insertion order).
* Indexed (supports positive & negative index).
* Mutable (can update/change).
* Duplicates allowed.
* Heterogeneous allowed.
* Dynamic in nature.

**🔹 3. Creating a List**

**Empty list**

a = []

print(a) # []

**From user input**

l = eval(input("Enter a list: "))

print(l)

print(type(l))

**Output**

Enter a list: [10,23,45,63,41]

[10, 23, 45, 63, 41]

<class 'list'>

**From range**

l = list(range(10, 23))

print(l) # [10, 11, 12, ..., 22]

**From string**

l = list("yuresh")

print(l) # ['y', 'u', 'r', 'e', 's', 'h']

print(type(l)) # <class 'list'>

**4. Accessing Elements of List**

There are **two methods**:

1. **Indexing**
2. **Slicing**

l = [10, 43, 34, 34, 34]

print(l[4]) # 34 (positive index)

print(l[-2]) # 34 (negative index)

print(l[5::-2]) # slicing with step

**Mutability**

print(id(l)) # memory address

l[0] = 12

print(id(l)) # same id → proves mutability

print(l)

**🔹 5. Traversing a List**

**Using while loop**

l = [12, 34, 23, 5, 3, 23]

i = 0

while i < len(l):

print(l[i])

i += 1

**Using for loop**

for x in l:

print(x)

**🔹 6. Important Functions of List**

l = [12, 34, 23, 5, 3, 23]

print(len(l)) # 6 → total elements

print(l.count(10)) # 0 → count of 10

print(l.index(3)) # 4 → index of element 3

**🔹 7. Manipulating Elements**

**Adding**

l = [12, 34, 23, 5, 3, 23]

l.append(42)

l.append("ram")

print(l)

**User input + even numbers**

l = []

for i in range(50, 101):

if i % 2 == 0:

l.append(i)

print(l) # even numbers from 50 to 100

**Insert**

l = [12, 34, 23, 5, 3, 23]

l.insert(1, 788) # insert at index 1

print(l)

**Update**

l[1] = 23

print(l)

**🔹 8. Extend Function**

* append() → adds one element.
* extend() → adds all elements of another list/string.

l = [12, 34, 23, 5, 3, 23]

k = ["ram", "radhe", "radhe"]

l.extend(k) # add elements of k

# l.append(k) # adds k as single element

# l.extend("ram") # adds each char

l.remove("ram")

print(l)

**🔹 9. Removing Elements**

**Remove all repetition of a value**

l = [12, 34, 23, 5, 3, 23, 34, 35, 34]

x = int(input("Enter a value to remove: "))

while True:

if x in l:

l.remove(x)

else:

break

print(l)

**pop()**

l = [12, 34, 23, 5, 3, 23, 34, 35, 34]

l.pop() # removes last element

l.pop(4) # removes element at index 4

print(l)

**🔹 10. Ordering Elements**

**Reverse**

l.reverse()

print(l)

**Sort**

l = [12, 34, 23, 5, 3, 23, 34, 35, 34]

l.sort()

print(l) # ascending

l.reverse()

print(l) # descending

**🔹 11. Aliasing and Cloning**

**Aliasing**

l = [12, 34, 23]

m = l

print(id(l), id(m)) # same id

m[1] = 12

print(m)

print(l) # both affected

**Cloning**

l = [12, 34, 23]

m = l[:] # or l.copy()

print(id(l), id(m)) # different ids

m[1] = 99

print(m)

print(l) # original unchanged

**🔹 12. Other Operations**

x = [12, 34, 54, 64]

y = [45, 56, 78, 23]

print(x + y) # Concatenation

print(x + [67]) # Adding new element

print(x \* 2) # Repetition

print(x == y) # Comparison

**Membership**

x = [23, 54, 454, 34, 34]

print(23 in x) # True

**Nested list access**

x = [23, 54, 454, 34, 34, [23, 45, 34]]

print(x[5][2]) # 34

**🔹 13. List Comprehension (VVV Imp)**

**📌 Definition:**

List comprehension is a concise and powerful way to create lists in Python using a single line of code.  
It combines looping + condition + expression inside square brackets [].

📌 General Syntax:

[expression for item in iterable if condition]

* expression → operation/output to be stored in the list
* item → variable that takes values from iterable (like for loop)
* iterable → sequence (list, range, string, etc.)
* condition (optional) → filter (only items satisfying condition are added)

📌 Example 1: Simple List Comprehension

numbers = [i for i in range(1, 6)]

print(numbers)

✅ Output:

[1, 2, 3, 4, 5]

(Same as writing a loop and appending each element.)

📌 Example 2: Square of Numbers

squares = [i\*i for i in range(1, 6)]

print(squares)

✅ Output:

[1, 4, 9, 16, 25]

Example 3: With Condition (Even Numbers Only)

evens = [i for i in range(1, 11) if i % 2 == 0]

print(evens)

✅ Output:

[2, 4, 6, 8, 10]

📌 Example 4: Extract First Letter of Each Word

words = ['Ram', 'Shyam', 'Hari']

first\_letters = [w[0] for w in words]

print(first\_letters)

✅ Output:

['R', 'S', 'H']

📌 Example 5: Nested List Comprehension (Matrix Flatten)

matrix = [[1, 2], [3, 4], [5, 6]]

flat = [num for row in matrix for num in row]

print(flat)

✅ Output:

[1, 2, 3, 4, 5, 6]

📌 Example 6: Uppercase and Length of Words

sentence = "my name is Ram Prasad".split()

result = [[word.upper(), len(word)] for word in sentence]

print(result)

✅ Output:

[['MY', 2], ['NAME', 4], ['IS', 2], ['RAM', 3], ['PRASAD', 6]]

📌 Advantages of List Comprehension:

1. Short and readable code.
2. Faster than normal loops with append().
3. Can include conditions and nested loops.

**🔹 15. Built-in Functions with Lists**

* len(l) → number of elements
* min(l) → smallest element
* max(l) → largest element
* sum(l) → sum of elements (numeric only)
* any(l) → True if at least one element is True
* all(l) → True if all elements are True

nums = [1, 2, 3, 4]

print(len(nums)) # 4

print(sum(nums)) # 10

print(min(nums)) # 1

print(max(nums)) # 4

print(any(nums)) # True

print(all(nums)) # True

**✅ Final Summary**

* **List** = Ordered, mutable, heterogeneous, dynamic collection.
* Allows **duplication, indexing, slicing, traversal, and comprehensions**.
* Provides **powerful methods**: append(), insert(), extend(), remove(), pop(), clear(), sort(), reverse().
* Supports **aliasing, cloning, concatenation, repetition, membership, nested lists**.
* **List comprehensions** make list creation & filtering easier.
* Very important in **real-life applications, data storage, and algorithms**.