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** 🔍

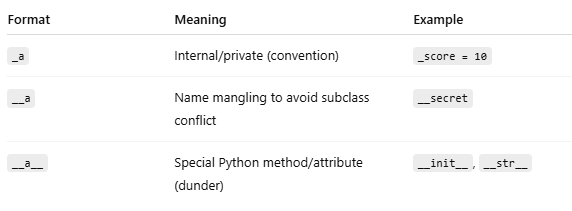
* Different cases help programmers **know what the variable represents**:
  + snake\_case → regular variable or function
  + PascalCase → class name
  + UPPER\_CASE → constant value

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 |



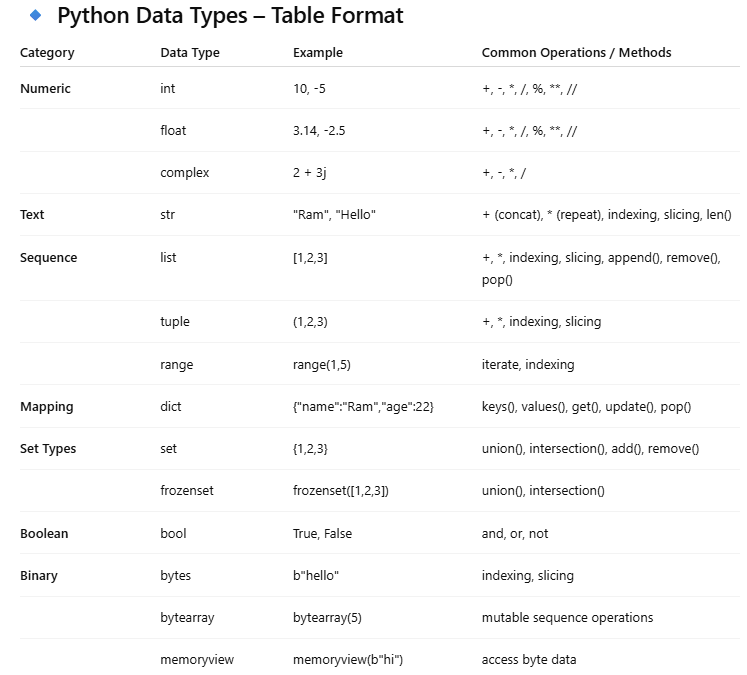
Paxadi ramro snaga details ma xa

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+)**

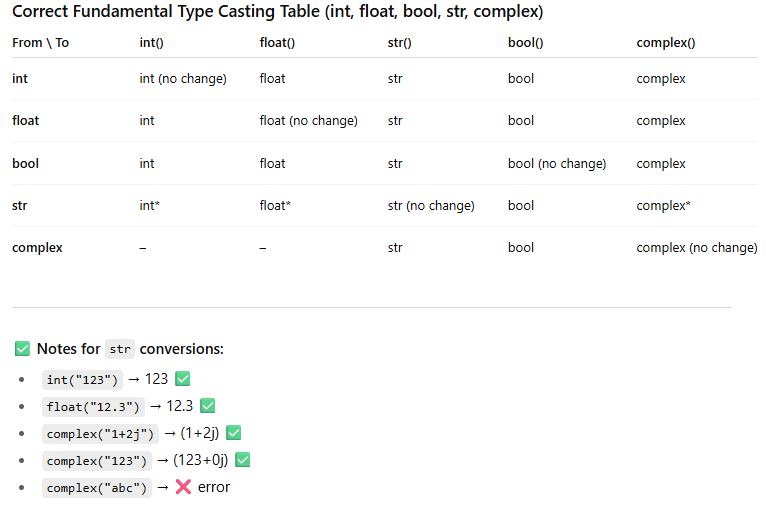
name = "Ram"

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

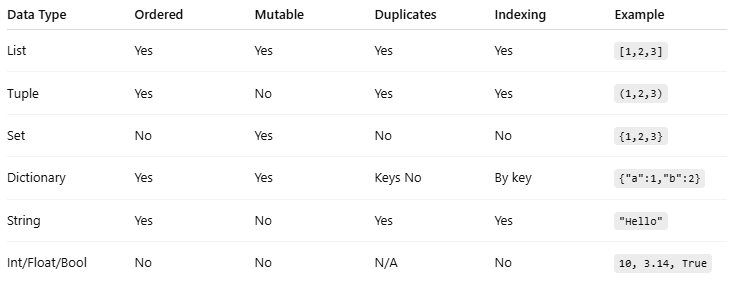
print(id(x)) # address of 10

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"])

}

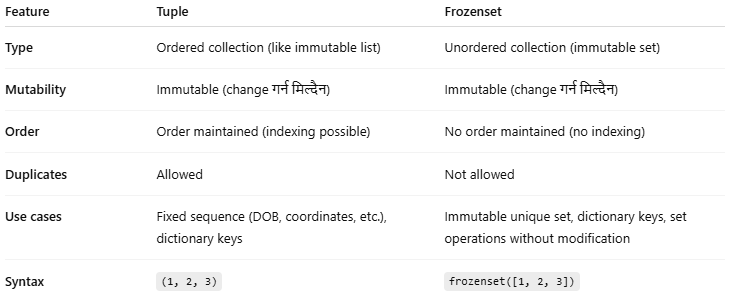
print(graph["A"]) # frozenset({'B', 'C'})

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:

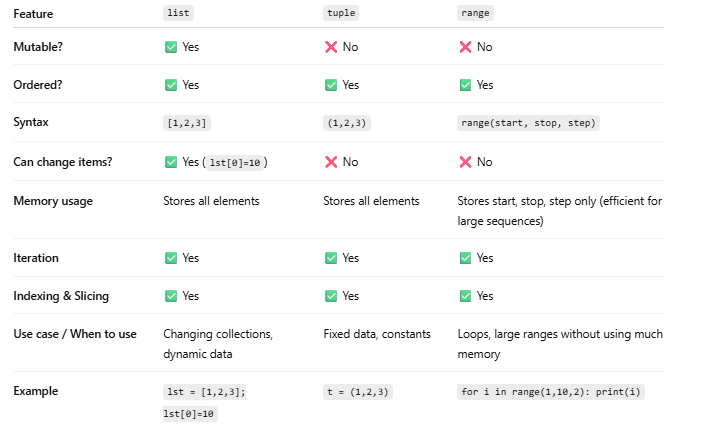
b1 = b'hi'

b2 = b'bye'

print(b1 + b2) # b'hibye'

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





**Day 5:**

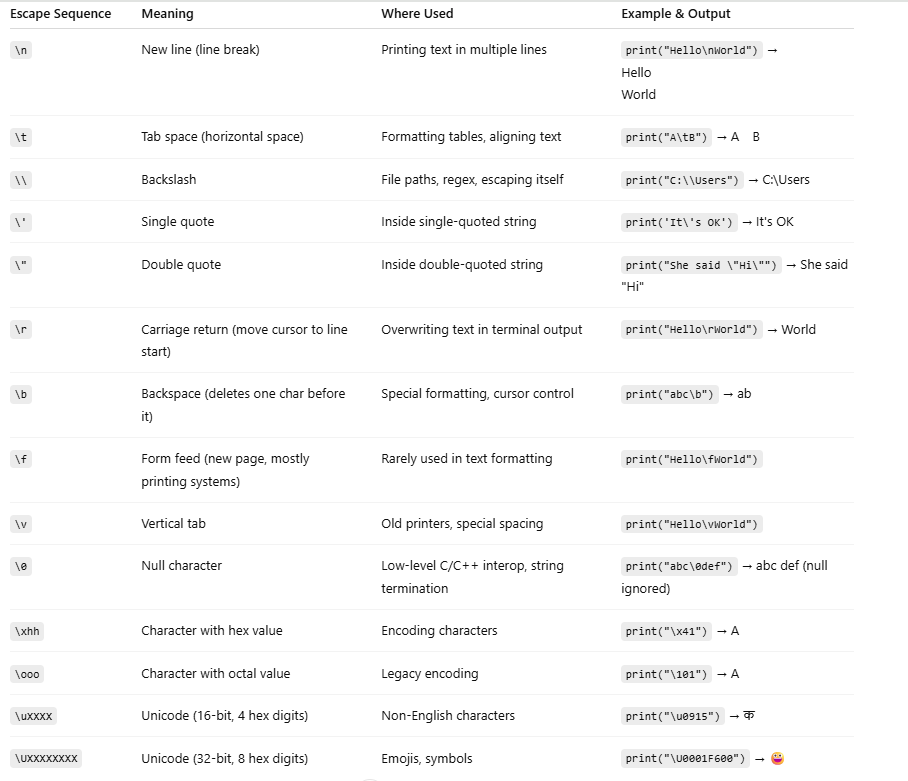
Python Escape Sequences

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]

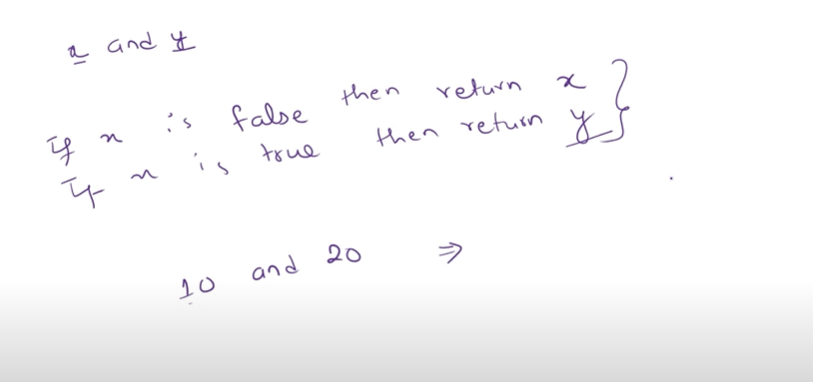
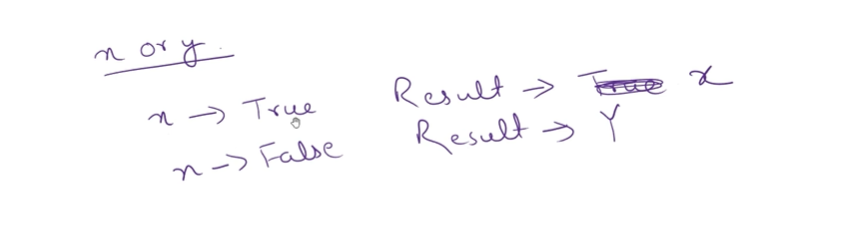
b = [1,2]

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

