# BASIC PYTHON NOTES

Day 1: Introduction to Python and Installation

## 🐍 Introduction to Python and Installation

### 🔹 1. What is Python?

**Python** is a **high-level**, **interpreted**, and **general-purpose programming language** known for its **simplicity and readability**.
It is widely used in many fields — from **data analysis, web development, AI/ML, automation**, to **game development**.

✅ Example:

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

**Output:**

```
Hello, World!
```

🔸 This one simple line is a complete program in Python — no need for curly braces, semicolons, or complex syntax.

### 🔹 2. History of Python

| Year | Event |
| --- | --- |
| **1989** | Python was conceived by **Guido van Rossum** in the Netherlands. |
| **1991** | Python 0.9.0 was released (first public release). |
| **2000** | Python 2.0 introduced (major update). |
| **2008** | Python 3.0 released (modern version). |
| **Today** | Latest stable version: **Python 3.x** (actively maintained). |

⚡ The name “Python” was inspired by the comedy show **“Monty Python’s Flying Circus”**, not the snake!

## 🔹 3. Why Python? (Key Features)

| Feature | Description |
| --- | --- |
| 🧩 **Simple and Easy to Learn** | Python’s syntax is very close to English. |
| ⚙️ **Interpreted Language** | Code executes line-by-line — no need for compilation. |
| 📚 **Huge Standard Library** | Ready-to-use modules for math, files, web, etc. |
| 🧠 **Versatile** | Can be used for Web, AI, ML, Data Science, Automation, etc. |
| 🔄 **Cross-Platform** | Runs on Windows, macOS, and Linux without modification. |
| 👥 **Large Community** | Millions of developers and tons of free resources. |

## 🔹 4. Python Uses in Real-World

| Field | Example |
|---|---|
| 🧮 **Data Science** | Data cleaning, analysis, visualization (Pandas, NumPy, Matplotlib) |
| 🤖 **Machine Learning / AI** | TensorFlow, Scikit-learn |
| 🌐 **Web Development** | Django, Flask |
| ⚙️ **Automation / Scripting** | Automate tasks (file handling, data entry, etc.) |
| 🧰 **Software Testing** | Selenium, PyTest |
| 🎮 **Game Development** | Pygame |
| 🕵️ **Cybersecurity** | Ethical hacking, scripting |
| 💬 **Chatbots & AI Assistants** | NLP libraries, OpenAI API |

## 🔹 5. How Python Works (Execution Flow)

### 🧠 Step-by-Step:

1. **Write Python code** → saved as .py file
   Example: hello.py
2. **Python Interpreter reads** code line by line.
3. It converts it into **bytecode** (intermediate form).
4. **Python Virtual Machine (PVM)** executes the bytecode.

### ⚙️ Diagram (conceptually):

Your Code (.py)

↓

Python Interpreter

↓

Bytecode (.pyc)

↓

Python Virtual Machine (PVM)

↓

Output

## 🔹 6. Installing Python (Step-by-Step)

### 🖥️ Step 1: Download Python

- Go to the official website: [https://www.python.org/downloads/](https://www.python.org/downloads/)
- Click on **“Download Python 3.x.x”** (latest version).

### 🧰 Step 2: Run Installer

When you open the downloaded file (python-3.x.x.exe):

✅ **Important:**
✔️ Check the box **“Add Python to PATH”**
✔️ Click **Install Now**

This ensures that you can run Python commands directly from the terminal or command prompt.

### 🪟 Step 3: Verify Installation

Open **Command Prompt (Windows)** or **Terminal (Mac/Linux)** and type:

python --version

**Output:**

Python 3.x.x

## 🔹 7. Writing and Running Your First Python Program

### Option 1: Using IDLE (comes with Python)

1. Open **IDLE** (search “IDLE” in start menu).
2. Type:
3. print("Hello, Naveen! Welcome to Python.")
4. Press **Enter** — you’ll see:
5. Hello, Naveen! Welcome to Python.

### Option 2: Using Command Prompt / Terminal

1. Open **Notepad**, type:
2. print("Hello Python!")
3. Save file as **hello.py**
4. Run it in command prompt:
5. python hello.py
6. Output:
7. Hello Python!

### Option 3: Using VS Code (Recommended for Projects)

1. Install **VS Code** from https://code.visualstudio.com/.
2. Install the **Python Extension**.
3. Open your .py file.
4. Click ▶ **Run Code** or press **Ctrl + F5**.

## 🔹 8. Python IDEs and Editors

| Tool | Description | Best For |
|---|---|---|
| 🟦 **IDLE** | Default Python IDE | Beginners |
| 🟨 **VS Code** | Lightweight, extensible editor | Projects, Coding |
| 🟥 **PyCharm** | Full-featured IDE | Professional Developers |
| 🟩 **Jupyter Notebook** | Web-based tool | Data Science, AI/ML |
| 🟧 **Spyder** | Scientific Python IDE | Data Analysis |

## 🔹 9. Python File Extension and Comments

| Type | Example | Description |
| --- | --- | --- |
| 🗂️ File Extension | example.py | Python files always end with .py |
| 💬 Single-line Comment | # This is a comment | Ignored by interpreter |
| 💬 Multi-line Comment | ''' This is multi-line comment ''' | Ignored by interpreter |

Example:

```
# This program adds two numbers

a = 10

b = 20

print(a + b)   # Output: 30
```

## 🔹 10. Understanding Interactive Mode vs Script Mode

| Mode | Description | Example |
| --- | --- | --- |
| **Interactive Mode** | Executes one line at a time directly in shell (REPL) | Use >>> prompt |
| **Script Mode** | Write full program in .py file and run | Example: python hello.py |

Example:

```
>>> print("Hi Naveen")

Hi Naveen

vs

# hello.py

print("Hi Naveen")
```

Run: python hello.py

## 🔹 11. Python Versions (2 vs 3)

| Feature | Python 2 | Python 3 |
|---|---|---|
| Status | Discontinued | Active |
| Print Syntax | print "Hello" | print("Hello") |
| Unicode Support | Limited | Full Unicode Support |
| Division | Integer by default | True division |
| Recommended? | ❌ No | ✅ Yes |

👉 **Always use Python 3** (currently version 3.13 as of 2025).

## 🔹 12. Checking Installed Libraries

To check which libraries are installed:

pip list

To install a new package (e.g., pandas):

pip install pandas

## 🔹 13. Environment Variables (PATH)

When you check **"Add Python to PATH"**, it tells your system **where to find the Python executable** — so you can run Python anywhere in your terminal.

If you forgot to check it during installation:

- Open **System Environment Variables**
- Edit **PATH**
- Add the folder path:
  Example: C:\Users\Naveen\AppData\Local\Programs\Python\Python311\

## 🔹 14. Upgrading Python

To upgrade to the latest version:

pip install --upgrade python

Or visit python.org/downloads.

## 🔹 15. Summary

| Topic | Key Point |
|---|---|
| Python | High-level, interpreted language |
| Creator | Guido van Rossum |
| File Extension | .py |
| Execution | Interpreted line-by-line |
| Latest Version | Python 3.x |
| Tool to Install Packages | pip |
| Run Program | python filename.py |
| IDEs | IDLE, VS Code, PyCharm, Jupyter |

## ✅ Example Practice for You

Create a file named intro.py and write:

```
# Welcome message

print("Hi Naveen, welcome to Python world!")

print("Let's learn Python step by step.")

print("Python version check successful!")
```

Run:

```
python intro.py
```

**Output:**

```
Hi Naveen, welcome to Python world!
```

Let's learn Python step by step.

Python version check successful!

Day 2: Python Syntax and First Program

# 🐍 Day 2: Python Syntax and First Program

## 🔹 1. What is Python Syntax?

**Python Syntax** refers to the **set of rules** that defines how a Python program is written and interpreted.
Just like grammar in English, **syntax** in Python defines **how to structure your code** so that the computer can understand it.

🧠 Think of syntax as the "grammar" of the Python language.

## 🔹 2. Python's Design Philosophy

Python is built around the **"Zen of Python"** — a collection of guiding principles.

Run this in Python:

import this

You'll see:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Readability counts.

✅ Meaning:
Python focuses on **clarity, simplicity, and readability** — making it beginner-friendly and efficient.

## 🔹 3. Writing Python Code

Python programs are usually saved in files ending with the extension:

filename.py

Example:

hello.py

Then run it using:

python hello.py

## 🔷 4. Python Syntax Rules (Most Important Basics)

Let’s go step-by-step 👇

### 🧩 Rule 1: Indentation (VERY IMPORTANT)

Unlike other languages (like C, Java, etc.), Python **uses indentation (spaces or tabs)** to define code blocks — not curly braces {}.

✅ Correct:

```
if True:
    print("This is correct indentation")
```

❌ Incorrect:

```
if True:
print("This will cause an IndentationError")
```

💡 **Rule:** Use **4 spaces** for indentation — it's the standard in Python.

### 🧩 Rule 2: Case Sensitivity

Python is **case-sensitive**, meaning variable names with different cases are treated differently.

Example:

```
name = "Naveen"
Name = "Python"
print(name)
print(Name)
```

**Output:**

```
Naveen
```

Python

✅ name and Name are two different variables.

## 🧩 Rule 3: Comments in Python

Comments are lines ignored by the interpreter — used to explain your code.

Single-line Comment:

# This is a single-line comment

print("Learning Python")

Multi-line Comment:

'''

This is a

multi-line

comment

'''

print("Hello Naveen")

## 🧩 Rule 4: Print Statements

print() is used to display output on the screen.

Example:

print("Welcome to Python!")

You can print multiple items using commas:

print("Name:", "Naveen", "Language:", "Python")

**Output:**

Name: Naveen Language: Python

## 🧩 Rule 5: Variables and Assignment

You don't need to declare a variable's type — Python automatically detects it.

Example:

x = 10

```
name = "Naveen"

pi = 3.14

print(x, name, pi)
```

**Output:**

```
10 Naveen 3.14
```

⚡ Variables are **created when you assign a value** to them — no need for a var or int keyword.

## 🧩 Rule 6: Line Breaks and Continuation

Each statement ends at the end of a line.
If a statement is too long, you can split it using a backslash \.

Example:

```
total = 10 + 20 + 30 + \
        40 + 50

print(total)
```

Or, within parentheses ():

```
total = (10 + 20 + 30 +
         40 + 50)

print(total)
```

## 🧩 Rule 7: Quotation Marks for Strings

Python supports **single (' ')**, **double (" ")**, and **triple quotes (''' ''' or """ """)**.

Example:

```
string1 = 'Hello'

string2 = "World"

string3 = '''Welcome to

Python'''

print(string1)

print(string2)
```

print(string3)

## 🧩 Rule 8: Multiple Statements on One Line

You can write more than one statement on the same line using a semicolon ;

Example:

a = 10; b = 20; print(a + b)

Output:

30

⚠️ Not recommended for readability — better write one statement per line.

## 🧩 Rule 9: Input from User

To take input from the user, use input() function.

Example:

name = input("Enter your name: ")

print("Welcome,", name)

**Output:**

Enter your name: Naveen

Welcome, Naveen

## 🧩 Rule 10: Keywords

Python has **reserved words** that cannot be used as variable names.

Example:

False, None, True, and, as, assert, 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

You can view all keywords using:

import keyword

print(keyword.kwlist)

## 🔹 5. Your First Python Program 🧠

Let’s create your **first real Python file** step by step.

### Step — Open any text editor (IDLE or VS Code)

Create a new file named first_program.py.

### Step — Type the following:

```
# My First Python Program


print("Hello Naveen!")

print("Welcome to the world of Python.")

print("Let's start coding step by step.")
```

### Step — Save and Run

In terminal or command prompt:

```
python first_program.py
```

### ✅ Output:

```
Hello Naveen!

Welcome to the world of Python.

Let's start coding step by step.
```

🎉 Congratulations! You just wrote and executed your **first Python program**!

## 🔹 6. Understanding the Program Line-by-Line

```
# My First Python Program
```

🟢 This is a comment — it is ignored by Python.

```
print("Hello Naveen!")
```

🟢 The print() function displays text on the screen.

```
print("Welcome to the world of Python.")
```

🟢 Every print statement executes line-by-line, showing the message in order.

## 🔹 7. Errors and Debugging Basics

Even beginners face small syntax errors. Let's see common ones:

| Error | Cause | Example |
|---|---|---|
| SyntaxError | Wrong syntax | print "Hello" ❌ |
| IndentationError | Missing indentation | if True: print("Hi") ❌ |
| NameError | Variable not defined | print(x) ❌ |
| TypeError | Wrong data type | "2" + 3 ❌ |

✅ Correct usage:

```
print("Hello")

if True:
    print("Indented properly")
```

## 🔹 8. Comments, Spacing, and Readability

Python's **PEP 8** style guide promotes readability.

**Good Example ✅**

```
name = "Naveen"

age = 25

print(name, age)
```

**Bad Example ❌**

```
name="Naveen";age=25;print(name,age)
```

🧠 Always write **clean, readable, and well-indented** code — it's not just for others but for your future self.

## 🔹 9. Interactive Mode Example

Open your **Python shell** (IDLE or terminal):

```
>>> print("Python is fun!")
```

```
Python is fun!
```

```
>>> 5 + 10
```

```
15
```

```
>>> name = "Naveen"
```

```
>>> print(name)
```

```
Naveen
```

💬 Interactive mode is great for quick testing.

## 🔹 10. Summary Table

| Concept | Description | Example |
|---|---|---|
| **Syntax** | Rules to write Python code | Indentation, keywords, etc. |
| **Comments** | Notes for humans | # This is a comment |
| **Indentation** | Defines code blocks | if True: print("Yes") |
| **Variables** | Store data | x = 10 |
| **Printing Output** | Show result on screen | print("Hi") |
| **Input** | Take user input | name = input() |
| **Errors** | Bugs in code | SyntaxError, NameError |

## 🔹 11. Practice Task for You 🧠💪

Create a file named intro_task.py with:

```
# Task 1: Greeting Program

name = input("Enter your name: ")
```

```
language = input("Which programming language are you learning? ")

print("Hello", name + "!")

print("Great to know that you are learning", language)
```

**Output:**

```
Enter your name: Naveen

Which programming language are you learning? Python

Hello Naveen!

Great to know that you are learning Python
```

## 🔹 12. Recap

✅ Python syntax = simple, clear, readable
✅ Indentation defines blocks
✅ Use print() to show output
✅ Use input() to take user input
✅ Every .py file is a script
✅ Errors = good learning signs

Day 3: Variables and Data Types

# 🐍 Day 3: Python Variables and Data Types

## 🔹 1. What is a Variable?

A **variable** is a **name** that stores a value in memory.
You can think of it as a **container** or a **label** used to store data temporarily while a program runs.

🧠 **Think like this:**

A variable is like a storage box with a name on it — you can put data inside, change it later, or take it out when needed.

✅ Example:

```
name = "Naveen"

age = 25

language = "Python"


print(name)

print(age)

print(language)
```

**Output:**

```
Naveen

25

Python
```

Here:

- name → variable storing a string "Naveen"
- age → variable storing a number 25
- language → variable storing "Python"

## 🔹 2. Rules for Defining Variables

Python has a few simple but strict rules for naming variables:

| Rule | Example | Valid / Invalid |
|---|---|---|
| Must start with a letter or underscore _ | _value = 10 | ✅ Valid |
| Cannot start with a number | 2name = "Naveen" | ❌ Invalid |
| Can only contain letters, digits, and underscores | user_name1 = "Naveen" | ✅ Valid |
| Case-sensitive | Age and age are different | ✅ Valid |
| Cannot use Python keywords | class = "A" | ❌ Invalid |

✅ Examples of valid variable names:

userName = "Naveen"

_user = 5

user_123 = "Data"

❌ Examples of invalid variable names:

123user = "Python"

for = 10

user name = "Naveen"

## 🔹 3. Dynamic Typing in Python

In Python, you **don't need to declare the data type** of a variable.
The interpreter automatically assigns it based on the value.

Example:

```
x = 10      # int
x = "Hello"  # str
x = 3.5      # float
```

✅ Python automatically understands the type of x based on what value you assign.

## 🔹 4. Variable Reassignment

You can **change the value** of a variable anytime — even to a different type.

Example:

```
a = 10
print(a)


a = "Naveen"
print(a)
```

**Output:**

10

Naveen

💡 This flexibility makes Python **dynamically typed**.

## 🔹 5. Multiple Variable Assignment

Python allows multiple assignments in one line.

### Example 1: Assigning multiple values

x, y, z = 10, 20, 30

print(x, y, z)

**Output:**

10 20 30

### Example 2: Assigning same value to multiple variables

a = b = c = "Python"

print(a, b, c)

**Output:**

Python Python Python

## 🔹 6. Deleting Variables

You can delete a variable using the del keyword.

Example:

x = 100

del x

print(x)

**Output:**

NameError: name 'x' is not defined

## 🔹 7. Constants in Python

Python does **not** have true constants,
but by convention, variables written in **UPPERCASE** are treated as constants.

Example:

PI = 3.14159

GRAVITY = 9.8

print(PI, GRAVITY)

**Output:**

3.14159 9.8

💡 Python doesn't prevent changing them, but developers treat them as fixed values.

## 🔹 8. What are Data Types?

A **data type** defines the kind of value a variable holds.

Python has several built-in data types grouped as follows:

| Category | Data Types |
|---|---|
| **Basic (Scalar)** | int, float, complex, bool, str |
| **Sequence** | list, tuple, range |
| **Mapping** | dict |
| **Set** | set, frozenset |
| **Binary** | bytes, bytearray, memoryview |
| **None Type** | None |

## 🔹 9. Basic Data Types Explained with Examples

### 🧮 1. Integer (int)

Whole numbers — positive, negative, or zero.

a = 10

```
b = -25

c = 0

print(type(a))
```

**Output:**

```
<class 'int'>
```

## 💧 2. Float (float)

Numbers with decimals.

```
x = 10.5

y = -3.14

print(type(x))
```

**Output:**

```
<class 'float'>
```

## 🔢 3. Complex (complex)

Numbers with a real and imaginary part.

```
num = 2 + 3j

print(num.real)

print(num.imag)
```

**Output:**

```
2.0

3.0
```

## 🔠 4. String (str)

Sequence of characters enclosed in quotes.

```
name = "Naveen"

print(name)

print(type(name))
```

**Output:**

Naveen

<class 'str'>

You can use:

- Single quotes ' '
- Double quotes " "
- Triple quotes ''' ''' for multi-line strings.

### ✅ 5. Boolean (bool)

Stores True or False values.

is_coding_fun = True

is_sun_cold = False

print(is_coding_fun)

print(type(is_sun_cold))

**Output:**

True

<class 'bool'>

💡 Used in conditional statements like if, while, etc.

### 💎 6. None Type (None)

Represents an **absence of value**.

data = None

print(data)

print(type(data))

**Output:**

None

<class 'NoneType'>

## 🔹 10. Type Checking

You can check the type of any variable using:

x = 100

print(type(x))

You can also use isinstance():

print(isinstance(x, int))

Output:

<class 'int'>

True

## 🔹 11. Type Casting (Changing Data Type)

You can **convert one data type into another** using casting functions.

| Conversion | Function | Example |
|---|---|---|
| To Integer | int() | int(3.14) → 3 |
| To Float | float() | float(5) → 5.0 |
| To String | str() | str(25) → '25' |
| To Boolean | bool() | bool(0) → False |

### Example:

x = 10

y = 3.5

z = "25"

print(float(x))  # int → float

print(int(y))  # float → int

print(int(z))  # string → int

print(str(x))  # int → string

**Output:**

10.0

3

25

'10'

## 🔹 12. Type Casting in Real-Time

Example:

num1 = input("Enter number 1: ")

num2 = input("Enter number 2: ")

sum = num1 + num2

print(sum)

**Input:**

10

20

**Output:**

1020

(❌ Because input returns string values)

✅ Correct way:

sum = int(num1) + int(num2)

print(sum)

**Output:**

30

## 🔹 13. Memory Concept (Visual)

Think of variables as labeled boxes stored in memory:

+------------------+

| name | "Naveen" |

+------------------+

| age | 25 |

+------------------+

| pi | 3.14 |

+------------------+

Python automatically manages where data is stored — you just use the names!

## 🔹 14. Getting User Input and Checking Type

```
name = input("Enter your name: ")

age = input("Enter your age: ")


print("Name:", name)

print("Age:", age)

print("Data type of age:", type(age))
```

**Output:**

Enter your name: Naveen

Enter your age: 25

Name: Naveen

Age: 25

Data type of age: <class 'str'>

✅ Convert manually:

```
age = int(age)

print(type(age))  # Now int
```

## 🔹 15. Summary Table

| Concept | Description | Example |
|---|---|---|
| **Variable** | Named storage in memory | x = 10 |
| **Dynamic Typing** | Type auto-detected | x = "Hi" |
| **Reassignment** | Change variable type | x = 5 → x = "Five" |
| **Constant** | Uppercase variable | PI = 3.14 |
| **int** | Whole numbers | 10 |
| **float** | Decimal numbers | 10.5 |
| **complex** | Real + imaginary | 2 + 3j |
| **str** | Text data | "Naveen" |
| **bool** | True/False | True |
| **NoneType** | No value | None |
| **Type Casting** | Convert data type | int("10") → 10 |

## 🔹 16. Practice Task for You 🧠💪

Create a file called variables_task.py and write:

# Task 1: Variable Creation

name = input("Enter your name: ")

age = int(input("Enter your age: "))

height = float(input("Enter your height (in cm): "))

is_student = True

# Display all values

```
print("\n--- User Information ---")
print("Name:", name)
print("Age:", age)
print("Height:", height)
print("Is Student:", is_student)
print("Data Type of each:")
print(type(name))
print(type(age))
print(type(height))
print(type(is_student))
```

✅ Output:

```
Enter your name: Naveen
Enter your age: 25
Enter your height (in cm): 175.5

--- User Information ---
Name: Naveen
Age: 25
Height: 175.5
Is Student: True
Data Type of each:
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

## 🔹 17. Recap (Quick Review)

✅ Variables = Containers for storing data
✅ Data Types = Define the kind of data stored
✅ Python is Dynamically Typed
✅ Use type() to check variable type
✅ Type Casting converts one type to another
✅ Use constants for fixed values (by convention)

Day 4: Type Casting

# 🧠 Day 4: Type Casting in Python

## 🎯 Goal:

To understand **how to convert one data type into another** in Python and why it's important.

## 🔹 What is Type Casting?

Type casting means **changing the data type** of a value or variable into another data type.

For example:

- Converting an integer into a string.
- Converting a string into a float.
- Converting a float into an integer, etc.

In simple terms, you are **"repackaging" data** into a different container.

### 💡 Example:

```
a = 10       # integer

b = float(a)  # type cast int to float

print(b)
```

📤 Output:

```
10.0
```

Here, we converted an **integer (10)** into a **float (10.0)**.

## 🔸 Why is Type Casting Important?

Because:

- Different data types can't always work together directly.
  Example: You can't add a number and a string directly.
- You may need to convert data when taking **input from the user** (which comes as a string by default).
- You may want to store or display data in different formats.

## 🔹 Types of Type Casting

Python supports two types:

**Implicit Type Casting (Automatic)**

**Explicit Type Casting (Manual)**

## 🧩 1. Implicit Type Casting (Automatic Conversion)

Python automatically converts smaller data types into larger ones to prevent data loss.

✅ Example:

```
x = 5      # int

y = 2.5    # float



z = x + y   # int + float = float

print(z)

print(type(z))
```

📤 Output:

```
7.5

<class 'float'>
```

🔹 Explanation:
Python automatically converted x (int) into float before addition.

## 🧩 2. Explicit Type Casting (Manual Conversion)

You **manually** convert data types using **built-in functions** like:

- int()
- float()
- str()
- bool()

- list()
- tuple()
- set()
- dict()

## 🔸 Common Type Casting Functions

| Function | Description | Example | Output |
|---|---|---|---|
| int(x) | Converts x to integer | int(3.8) | 3 |
| float(x) | Converts x to float | float(5) | 5.0 |
| str(x) | Converts x to string | str(10) | '10' |
| bool(x) | Converts x to Boolean | bool(0) | False |
| list(x) | Converts iterable to list | list((1,2,3)) | [1, 2, 3] |
| tuple(x) | Converts iterable to tuple | tuple([1,2,3]) | (1, 2, 3) |
| set(x) | Converts iterable to set | set([1,2,2,3]) | {1, 2, 3} |

## 🧠 Example 1: int() to float() and string

```
num = 25

print(float(num))  # Convert int to float

print(str(num))    # Convert int to string
```

📤 Output:

```
25.0

'25'
```

## 🧠 Example 2: string → int

```
a = "100"

b = int(a) + 50

print(b)
```

📤 Output:

```
150
```

🧩 Explanation:
String "100" was converted to integer 100 before addition.

## ⚠️ Common Error in Type Casting

If a string cannot be converted to a number, it causes an error.

❌ Example:

```
a = "Hello"

b = int(a)   # Error
```

📤 Output:

```
ValueError: invalid literal for int() with base 10: 'Hello'
```

✅ Rule:
Only numeric strings like "123" can be converted using int() or float().

## 🧠 Example 3: Input Conversion

When you take input using input(), it always comes as a string.

```
age = input("Enter your age: ")

print(type(age))
```

📤 Output:

```
<class 'str'>
```

✅ To use it as a number:

```
age = int(input("Enter your age: "))

print(age + 5)
```

📤 Output:

If you enter 20 → 25

### 🧠 Example 4: Convert between sequences

```
# List to tuple

list1 = [1, 2, 3]

tuple1 = tuple(list1)

print(tuple1)


# Tuple to set

set1 = set(tuple1)

print(set1)
```

📤 Output:

```
(1, 2, 3)

{1, 2, 3}
```

## ⚡ Real-Life Example: Type Casting in a Calculator

```
num1 = input("Enter first number: ")

num2 = input("Enter second number: ")


# Convert strings to float

result = float(num1) + float(num2)


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

📤 Output:

```
Enter first number: 12.5

Enter second number: 7.5

Sum: 20.0
```

## 🔍 Summary Table

| Concept | Description | Example |
|---|---|---|
| Implicit Casting | Python does automatically | 5 + 2.5 → 7.5 |
| Explicit Casting | You do manually | int("10") → 10 |
| Common Functions | int(), float(), str(), bool(), list(), tuple(), set() | Various |
| Input Handling | input() gives string → cast before arithmetic | int(input()) |

## 🧩 Small Practice Tasks

1. Convert "25.6" (string) into float and print its type.
2. Add integer 5 and string "10" using correct casting.
3. Convert list [1,2,3,3] into a set and print the result.
4. Take two inputs from the user, cast to integers, and print their sum.
5. Convert boolean True into integer and float.

Day 5: Input and Output Functions

# 🧠 Day 5: Input and Output Functions in Python

## 🎯 Goal:

To understand how to **take input** from users and **display output** in Python programs using built-in functions.

## 🔹 What are Input and Output Functions?

In Python,

- **Input Function (input())** → is used to **take information** from the user.
- **Output Function (print())** → is used to **display information** to the user.

They are the **most commonly used functions** when building interactive programs.

## 🧩 Output Function — print()

### ➤ Purpose:

Displays messages, variables, or results on the screen.

### 🧠 Syntax:

print(object(s), sep=' ', end='\n')

### 🟦 Parameters:

| Parameter | Description | Default |
|---|---|---|
| object(s) | The text or variable to print | Required |
| sep | Separator between multiple objects | ' ' (space) |
| end | What to print at the end | '\n' (new line) |

### 💡 Basic Example:

print("Hello, Naveen!")

print("Welcome to Python.")

### 📤 Output:

Hello, Naveen!

Welcome to Python.

### 💡 Printing Multiple Values:

name = "Naveen"

age = 22

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

### 📤 Output:

My name is Naveen and I am 22 years old.

👉 The print() function **automatically separates values** with a space using the sep parameter.

## 💡 Using sep Parameter

print("Python", "is", "fun", sep="-")

📤 **Output:**

Python-is-fun

🧩 Explanation:
The separator between each word is changed from a **space** to a **hyphen (-)**.

## 💡 Using end Parameter

print("Hello", end=" ")

print("World!")

📤 **Output:**

Hello World!

🧩 Explanation:
The end parameter changed the default new line (\n) to a space (" ").

## 💡 Printing Variables Together

x = 10

y = 20

print("Sum of", x, "and", y, "is", x + y)

📤 **Output:**

Sum of 10 and 20 is 30

## 💡 Formatted Output — f-Strings

**Introduced in Python 3.6**, f-strings make formatting cleaner and faster.

name = "Naveen"

marks = 95

print(f"Hello {name}, you scored {marks}% in Python!")

📤 **Output:**

Hello Naveen, you scored 95% in Python!

🧠 **Why f-strings?**
They are **faster** and **easier to read** than using commas or format().

### 💡 String Formatting using .format()

name = "Naveen"

city = "Vijayawada"

print("My name is {} and I live in {}.".format(name, city))

📤 **Output:**

My name is Naveen and I live in Vijayawada.

## 🧩 Input Function — input()

### ➤ Purpose:

Used to take **user input** (keyboard input) into a program.

### 🧠 Syntax:

variable_name = input("Message for the user: ")

🧩 The **message inside quotes** is optional — it helps you prompt the user.

### 💡 Example 1: Taking Simple Input

name = input("Enter your name: ")

print("Hello,", name)

📤 **Output:**

Enter your name: Naveen

Hello, Naveen

### 💡 Example 2: Taking Numeric Input

⚠️ Important:
By default, everything entered through input() is **string** type.

Example:

```
age = input("Enter your age: ")

print(type(age))
```

📤 **Output:**

```
<class 'str'>
```

✅ To use it as a number, you must **type cast** it.

```
age = int(input("Enter your age: "))

print("After 5 years, you will be", age + 5)
```

📤 **Output:**

```
Enter your age: 20

After 5 years, you will be 25
```

## 💡 Example 3: Taking Multiple Inputs

```
name, age = input("Enter your name and age (separated by space): ").split()

print("Name:", name)

print("Age:", age)
```

📤 **Output:**

```
Enter your name and age (separated by space): Naveen 21

Name: Naveen

Age: 21
```

🧩 Explanation:

- The split() function breaks the input based on spaces.
- You can assign each part to separate variables.

## 💡 Example 4: Multiple Inputs as Numbers

```
a, b = map(int, input("Enter two numbers: ").split())

print("Sum:", a + b)
```

📤 **Output:**

```
Enter two numbers: 5 10
```

Sum: 15

🧩 Explanation:
map(int, input().split()) → converts each input string into an integer.

## 🧮 Input and Output Combined Example

name = input("Enter your name: ")

math = float(input("Enter marks in Math: "))

science = float(input("Enter marks in Science: "))

avg = (math + science) / 2

print(f"{name}, your average score is {avg}")

📤 **Output:**

Enter your name: Naveen

Enter marks in Math: 90

Enter marks in Science: 95

Naveen, your average score is 92.5

## ⚡ Real-Life Example: Simple Calculator

num1 = float(input("Enter first number: "))

num2 = float(input("Enter second number: "))

print("Sum:", num1 + num2)

print("Difference:", num1 - num2)

print("Product:", num1 * num2)

print("Division:", num1 / num2)

📤 **Output:**

Enter first number: 8

Enter second number: 2

Sum: 10.0

Difference: 6.0

Product: 16.0

Division: 4.0

## 🧩 Summary Table

| Function | Description | Example | Output |
|---|---|---|---|
| print() | Displays output | print("Hello") | Hello |
| input() | Takes user input (string) | input("Enter name: ") | user input |
| f-string | Formatted output | f"{name} is {age} years old" | e.g. "Naveen is 22 years old" |
| .format() | Alternate string formatting | "{} {}".format('Hello', 'World')" | Hello World |
| split() | Split input values | "5 10".split() | ['5', '10'] |
| map() | Apply a function (e.g., int) to all input values | map(int, input().split()) | [5, 10] |

## 🧠 Practice Exercises

Write a program to take your name and age, and print a greeting message.
Take two numbers as input and print their sum, difference, and product.
Take 3 subject marks from the user and print the total and average.
Ask the user for their name and favorite color, then print a personalized message.
Take multiple space-separated numbers from the user and print their sum.

# Day 6: Comments in Python

# 🧠 Day 6: Comments in Python

## 🎯 Goal:

To understand what **comments** are, why they are important, and how to use **single-line** and **multi-line** comments effectively in Python.

## 🔹 What is a Comment?

A **comment** is a piece of text in your code that **Python ignores** when the program runs.
It's used to **explain code**, **make notes**, or **disable** a line temporarily for testing.

🧩 Think of comments like "sticky notes" on your code — they help **humans understand** what's happening, but Python **skips them**.

### 💡 Example:

```
# This is a comment

print("Hello, Naveen!")  # This prints a greeting
```

**📤 Output:**

```
Hello, Naveen!
```

🧠 Explanation:

- The line starting with # is a **comment**.
- Python **ignores** it and only executes the print() statement.

## 🔸 Why Are Comments Important?

✅ To make your code **easier to read and understand**
✅ To **document** what your code does
✅ To **help other programmers** understand your logic
✅ To **debug** or **temporarily disable** code during testing

## 🔹 Types of Comments in Python

There are **two main types** of comments:

**Single-line Comments**
**Multi-line Comments (Block Comments)**

## 🧩 Single-Line Comments

### ➤ Syntax:

# This is a single-line comment

Everything after the # symbol on that line is ignored by Python.

### 💡 Example 1:

# This program prints my name

print("Naveen")

**📤 Output:**

Naveen

### 💡 Example 2: Inline Comment

x = 10  # assigning 10 to variable x

print(x)

**📤 Output:**

10

🧩 Explanation:
The text after # is ignored — it's just for the programmer's understanding.

### 💡 Example 3: Disabling Code Temporarily

# print("This line is disabled")

print("This line runs")

**📤 Output:**

This line runs

✅ Useful when testing — you can disable lines without deleting them.

## 🧩 Multi-Line Comments (Block Comments)

There is **no special syntax** for multi-line comments in Python,
but there are **two common ways** to write them 👇

### 💬 Method 1: Using # symbol repeatedly

```
# This is a multi-line comment

# that explains what the program does.

# Each line starts with a #.

print("Python is amazing!")
```

📤 **Output:**

```
Python is amazing!
```

🧩 Explanation:
Each line starting with # is ignored by Python.

### 💬 Method 2: Using Triple Quotes (''' or """)

Triple quotes are usually used for **docstrings** (documentation),
but they can also be used for multi-line comments.

```
'''

This program shows

how to use multi-line comments

in Python.

'''

print("Welcome to Python!")
```

📤 **Output:**

```
Welcome to Python!
```

✅ Python ignores text inside triple quotes if it's not assigned to a variable.

### ⚠️ Note:

Triple-quoted comments are technically **multi-line strings**,
but since they're **not assigned to a variable**, Python ignores them — so we can use them like comments.

## 🧠 Example: Single-Line vs Multi-Line Comments

```
# Single-line comment

name = "Naveen"
```

'''

Multi-line comment:

The next line will print a message

using the variable name.

'''

print("Hello,", name)

📤 **Output:**

Hello, Naveen

## 🧩 Docstrings (Documentation Strings)

Docstrings are **special comments** used to describe **functions**, **classes**, or **modules**.
They are enclosed in triple quotes (""" or ''') and can be accessed using the .__doc__ attribute.

### 💡 Example:

def greet():

"""This function prints a greeting message."""

print("Hello, Naveen!")

print(greet.__doc__)

greet()

📤 **Output:**

This function prints a greeting message.

Hello, Naveen!

🧠 Explanation:

- The triple quotes inside the function are **docstrings**.
- They describe what the function does.
- They can be read by tools or other developers for documentation.

## ⚡ Real-Life Example: Using Comments for Explanation

```
# Program to calculate area of a rectangle

# Step 1: Take input from the user
length = float(input("Enter the length: "))
width = float(input("Enter the width: "))

# Step 2: Calculate area
area = length * width

# Step 3: Display result
print("Area of Rectangle =", area)
```

📤 **Output:**

```
Enter the length: 10
Enter the width: 5
Area of Rectangle = 50.0
```

🧠 Explanation:
Comments make it very clear **what each part of the code does** —
even a beginner can understand it!

## 🧩 Summary Table

| Type | Symbol / Syntax | Description | Example |
|---|---|---|---|
| Single-line Comment | # | Used for short notes or explanations | # Print name |
| Inline Comment | # (after code) | Written at end of a code line | x = 10 # variable |
| Multi-line Comment | '''...''' or multiple # | Used for long explanations | '''Multiple lines''' |
| Docstring | """..""" | Describes function, class, or module | def func(): """Docs""" |

## 🧠 Good Commenting Practices

✅ Write **meaningful and short** comments
✅ Keep comments **up to date** when changing code
✅ Avoid commenting every single line — use them **only where logic is not obvious**
✅ Use **docstrings** for documenting functions and modules
✅ Don't overuse triple-quoted comments unnecessarily

## 🧩 Practice Exercises

Write a Python program with single-line comments explaining each step.
Use a multi-line comment to describe what your program does.
Create a function and write a docstring explaining its purpose.
Temporarily disable a print statement using a comment.
Mix inline comments and normal comments to describe a short program.

## 💡 Example Challenge (Small Task)

```
# Write a program to calculate and display the square of a number

# Step 1: Take input

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



# Step 2: Calculate square
```

square = num ** 2

# Step 3: Display result

print(f"The square of {num} is {square}")

📤 **Output:**

Enter a number: 6

The square of 6 is 36

Day 7: Arithmetic Operators

# 🧠 Day 7: Arithmetic Operators in Python

## 🎯 Goal:

To understand **Arithmetic Operators** in Python — what they are, how they work, and how to use them with **examples and real-world scenarios**.

## 🔹 What are Operators?

Operators are **symbols** that perform **operations** on **variables and values**.

In simple words:

Operators are “tools” that tell Python **what kind of calculation or comparison** you want to perform.

## 🔹 What are Arithmetic Operators?

Arithmetic operators are used to perform **mathematical calculations** like addition, subtraction, multiplication, division, etc.

They work just like in real-world math — but with Python syntax.

## 🧩 Arithmetic Operators List

| Operator | Name | Example | Result |
|---|---|---|---|
| + | Addition | 10 + 5 | 15 |
| - | Subtraction | 10 - 5 | 5 |
| * | Multiplication | 10 * 5 | 50 |
| / | Division | 10 / 5 | 2.0 |
| // | Floor Division | 10 // 3 | 3 |
| % | Modulus (Remainder) | 10 % 3 | 1 |
| ** | Exponent (Power) | 2 ** 3 | 8 |

## 🧠 Addition Operator (+)

Used to **add two numbers** or **combine strings**.

### 💡 Example 1: Numbers

a = 10

b = 20

print(a + b)

📤 **Output:**

30

### 💡 Example 2: Strings

first = "Python "

second = "Rocks"

print(first + second)

📤 **Output:**

Python Rocks

🧠 This is called **string concatenation** — joining strings using +.

## 🧠 Subtraction Operator (-)

Used to **subtract** one value from another.

x = 15

y = 7

print(x - y)

📤 **Output:**

8

🧩 You can also use negative numbers:

print(-x)

📤 **Output:**

-15

## 🧠 Multiplication Operator (*)

Used to **multiply numbers**.

x = 6

y = 4

print(x * y)

📤 **Output:**

24

### 💡 Example 2: String Repetition

print("Hi! " * 3)

📤 **Output:**

Hi! Hi! Hi!

🧠 You can multiply strings by integers to repeat them.

## 🧠 Division Operator (/)

Used to divide one number by another.
⚠️ Always returns a **float** (decimal) value — even if numbers divide evenly.

x = 10

y = 2

print(x / y)

📤 **Output:**

5.0

### 💡 Example 2:

print(7 / 2)

📤 **Output:**

3.5

🧠 Division always returns a floating-point number (float).

## 🧠 Floor Division Operator (//)

Used to divide and **remove the decimal part** (returns only the whole number).

print(7 // 2)

📤 **Output:**

3

🧠 Floor division "floors" the result — rounds **down** to the nearest whole number.

### 💡 Example 2:

print(-7 // 2)

📤 **Output:**

-4

🧠 Because it **rounds toward negative infinity**, not zero.

## 🧠 Modulus Operator (%)

Used to find the **remainder** of a division.

print(10 % 3)

📤 **Output:**

1

### 💡 Example 2:

```
print(25 % 5)
```

📤 **Output:**

0

🧠 Useful in:

- Checking if a number is **even or odd**
- Finding repeating patterns

✅ Example:

```
num = 7

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

📤 **Output:**

Odd

## 🧠 Exponent Operator (**)

Used to find the **power** of a number.

```
print(2 ** 3)
```

📤 **Output:**

8

🧠 Equivalent to:

2 × 2 × 2 = 8

### 💡 Example 2: Square and Cube

```
print(5 ** 2)  # square
```

```
print(3 ** 3)  # cube
```

📤 **Output:**

```
25

27
```

## ⚡ Real-Life Example: Simple Calculator

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

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


print("Addition:", a + b)

print("Subtraction:", a - b)

print("Multiplication:", a * b)

print("Division:", a / b)

print("Floor Division:", a // b)

print("Modulus:", a % b)

print("Exponent:", a ** b)
```

📤 **Output:**

```
Enter first number: 5

Enter second number: 2

Addition: 7

Subtraction: 3

Multiplication: 10

Division: 2.5

Floor Division: 2

Modulus: 1

Exponent: 25
```

# 🧩 Operator Precedence (Order of Operations)

When multiple operators appear in one expression,
Python follows a specific **order** — called **precedence**.

### 🧠 Order (Highest → Lowest):

| Operator | Meaning | Example | Result |
|---|---|---|---|
| ** | Exponent | 2 ** 3 | 8 |
| *, /, //, % | Multiplication, Division, Floor, Modulus | 10 / 2 * 3 | 15.0 |
| +, - | Addition, Subtraction | 5 + 2 - 3 | 4 |

### 💡 Example: Understanding Precedence

result = 10 + 2 * 3

print(result)

📤 **Output:**

16

🧠 Explanation:

- Multiplication happens first → 2 * 3 = 6
- Then addition → 10 + 6 = 16

### 💡 Example: Using Parentheses ()

Parentheses **change** the order of evaluation.

result = (10 + 2) * 3

print(result)

📤 **Output:**

36

🧠 Parentheses are always evaluated first — they can **override precedence**.

## 🧩 Summary Table

| Operator | Description | Example | Output |
|---|---|---|---|
| + | Addition | 7 + 3 | 10 |
| - | Subtraction | 7 - 3 | 4 |
| * | Multiplication | 7 * 3 | 21 |
| / | Division | 7 / 2 | 3.5 |
| // | Floor Division | 7 // 2 | 3 |
| % | Modulus | 7 % 2 | 1 |
| ** | Exponentiation | 2 ** 3 | 8 |

## 🧠 Common Real-World Uses

| Use Case | Example | Operator |
|---|---|---|
| Find even/odd number | num % 2 == 0 | % |
| Calculate square root | num ** 0.5 | ** |
| Count installments | total // months | // |
| Get total price | price * quantity | * |
| Combine text | "Name: " + name | + |

## 🧩 Practice Tasks

Write a program to calculate the area and perimeter of a rectangle using arithmetic operators.

Take two numbers and display the result of all arithmetic operations.

Write a program that checks if a number is divisible by both 3 and 5.

Calculate the cube of a number entered by the user.

Find the remainder when a number is divided by another number.

Day 8: Assignment Operators

# 🧠 Day 8: Assignment Operators in Python

## 🎯 Goal:

To understand **what assignment operators are**, **how they work**, and **how to use them** in different real-life situations in Python.

## 🔹 What are Assignment Operators?

👉 **Assignment Operators** are used to **assign values** to variables.

The **basic idea**:
They **store data** or **update** existing data in variables.

🧩 Think of them like a **delivery person** putting a package (value) into a box (variable).

### 💬 Example:

x = 10

Here:

- x → variable (box)
- = → assignment operator
- 10 → value (package)

So it means: **“Put 10 inside x.”**

## 🔹 Types of Assignment Operators in Python

Python has several types of assignment operators that allow you to perform operations and assign values **in a single step**.

Let’s explore them one by one 👇

## 🔶 Simple Assignment (=)

It assigns the **right-hand value** to the **left-hand variable**.

💡 Example:

```
x = 5

y = "Naveen"
```

🧠 Explanation:

- x gets 5
- y gets the string "Naveen"

📤 **Output:**

```
x = 5

y = Naveen
```

✅ Used when you first create or initialize variables.

## 🔶 Add and Assign (+=)

It **adds** the right-hand value to the variable and **stores** the result back in the same variable.

💡 Example:

```
x = 10

x += 5   # same as x = x + 5

print(x)
```

📤 **Output:**

```
15
```

🧠 Explanation:
It **adds 5** to the current value of x (10),
and assigns the **new value (15)** back to x.

## 🔶 Subtract and Assign (-=)

It **subtracts** the right-hand value and stores the result back in the variable.

💡 Example:

```
x = 20

x -= 8   # same as x = x - 8

print(x)
```

📤 **Output:**

```
12
```

🧠 Explanation:
Subtracts 8 from 20 and assigns the result (12) to x.

## 🔸 Multiply and Assign (*=)

It **multiplies** the variable by the right-hand value and assigns the result back.

💡 Example:

```
x = 6

x *= 3   # same as x = x * 3

print(x)
```

📤 **Output:**

```
18
```

🧠 Explanation:
6 * 3 = 18 → stores 18 in x.

## 🔸 Divide and Assign (/=)

It **divides** the variable by the right-hand value and assigns the result back.

💡 Example:

```
x = 25

x /= 5   # same as x = x / 5

print(x)
```

📤 **Output:**

```
5.0
```

🧠 Explanation:
25 ÷ 5 = 5.0 → result becomes a **float** value.

## 🔶 Floor Divide and Assign (//=)

It divides the variable and **rounds down** the result to the nearest integer.

💡 Example:

```
x = 22

x //= 5   # same as x = x // 5

print(x)
```

📤 **Output:**

```
4
```

🧠 Explanation:
22 ÷ 5 = 4.4 → floor division gives 4.

## 🔶 Modulus and Assign (%=)

It gives the **remainder** of the division and stores it back in the variable.

💡 Example:

```
x = 19

x %= 4   # same as x = x % 4

print(x)
```

📤 **Output:**

```
3
```

🧠 Explanation:
19 ÷ 4 = 4 remainder 3 → result is 3.

## 🔶 Exponent and Assign (=)**

It **raises** the variable to the power of the right-hand value and assigns the result back.

💡 Example:

```
x = 3

x **= 4   # same as x = x ** 4

print(x)
```

📤 **Output:**

81

🧠 Explanation:
$3^4 = 81$ → stored in x.

## 🧠 All Assignment Operators Summary Table

| Operator | Example | Same As | Description |
|---|---|---|---|
| = | x = 10 | x = 10 | Assigns value |
| += | x += 3 | x = x + 3 | Add and assign |
| -= | x -= 3 | x = x - 3 | Subtract and assign |
| *= | x *= 3 | x = x * 3 | Multiply and assign |
| /= | x /= 3 | x = x / 3 | Divide and assign (float) |
| //= | x //= 3 | x = x // 3 | Floor divide and assign |
| %= | x %= 3 | x = x % 3 | Modulus and assign |
| **= | x **= 3 | x = x ** 3 | Exponentiate and assign |

## 💡 Example Program – Salary Update

Let's use multiple assignment operators in one small example 👇

```
# Employee salary calculation


salary = 50000     # base salary

bonus = 5000
```

```
tax = 2000


salary += bonus    # add bonus
salary -= tax      # subtract tax
salary *= 1.10     # 10% increment


print("Final Salary:", salary)
```

📤 **Output:**

```
Final Salary: 59400.0
```

🧠 Explanation:

- Start: 50000
- Add bonus: 55000
- Subtract tax: 53000
- Apply 10% raise: 53000 × 1.10 = 58300 (final result)

## 💡 Example 2 – Bank Balance Update

```
balance = 1000


balance += 200   # deposit ₹200
balance -= 50    # withdraw ₹50
balance *= 1.05  # 5% interest


print("Updated Balance:", balance)
```

📤 **Output:**

```
Updated Balance: 1210.0
```

✅ Real-life use of assignment operators for **incremental updates**.

## 🧩 Shortcut Trick

Assignment operators help **save time** and make code **cleaner**.

Instead of writing:

x = x + 10

You can write:

x += 10

✔ Easier to read
✔ Less code
✔ More professional

## ⚠ Common Mistake to Avoid

🚫 You can't write 10 = x
✅ Correct form: x = 10

Reason:
The **left-hand side** of = must always be a **variable**,
and the **right-hand side** must be a **value or expression**.

## 🧠 Practice Exercises

Write a program that takes a number and doubles it using *=.
Write a program that increases a student's marks by 10 using +=.
Use /= and //= to divide a value and observe the difference.
Calculate the remainder of a number using %=.
Build a small calculator using assignment operators.

## 💡 Example Challenge

```
# Program to demonstrate all assignment operators


x = 10

print("Initial x:", x)


x += 2

print("After += :", x)


x -= 4
```

```
print("After -= :", x)

x *= 3
print("After *= :", x)

x /= 2
print("After /= :", x)

x //= 2
print("After //= :", x)

x %= 3
print("After %= :", x)

x **= 2
print("After **= :", x)
```

📤 **Output:**

```
Initial x: 10
After += : 12
After -= : 8
After *= : 24
After /= : 12.0
After //= : 6.0
After %= : 0.0
After **= : 0.0
```

## 🧩 Summary Recap

- Assignment operators **store or update** values in variables.
- They make code **shorter, cleaner, and efficient**.
- Used widely in **loops, calculations, financial apps, and games**.
- Combine **operation + assignment** in one step.

Day 9: Comparison Operators

# 🧠 Day 9: Comparison (Relational) Operators in Python

## 🎯 Goal:

To learn how to **compare two values** using **Comparison Operators**, understand their **results (True or False)**, and use them in **real-world decision-making programs**.

## 🔹 What Are Comparison Operators?

**Comparison operators** are used to **compare two values** (numbers, strings, or variables).
They return a **Boolean result** — either True or False.

🧠 Think of it like **asking questions** to Python.

👉 Examples:

- "Is 10 greater than 5?" → ✅ **True**
- "Is 8 equal to 9?" → ❌ **False**

## 💡 Example:

```
x = 10

y = 5


print(x > y)
```

📤 **Output:**

```
True
```

🧠 Explanation:
Python checks if 10 > 5 — Yes, that's true.

## 🔶 Why Are Comparison Operators Important?

✅ Used in **conditions** (like if, while loops)
✅ Used in **decision-making** programs
✅ Helps compare **numbers, strings, or variables**
✅ Essential for **logic building** in coding

## 🔷 Types of Comparison Operators

There are **six main comparison operators** in Python:

| Operator | Meaning | Example | Result |
|---|---|---|---|
| == | Equal to | 10 == 10 | True |
| != | Not equal to | 10 != 5 | True |
| > | Greater than | 10 > 5 | True |
| < | Less than | 4 < 7 | True |
| >= | Greater than or equal to | 10 >= 10 | True |
| <= | Less than or equal to | 5 <= 8 | True |

Let's explore each one in detail 👇

## 🔶 Equal To (==)

It checks whether the **left value is equal to the right value**.

### 💡 Example:

x = 5

y = 5

```
print(x == y)
```

📤 **Output:**

```
True
```

🧠 Explanation:
Since x and y have the same value (5), the condition is True.

## 🔸 Not Equal To (!=)

It checks whether two values are **different**.

💡 Example:

```
a = 10

b = 7

print(a != b)
```

📤 **Output:**

```
True
```

🧠 Explanation:
10 is not equal to 7, so it returns True.

## 🔸 Greater Than (>)

Checks whether the **left value is greater than** the right value.

💡 Example:

```
x = 15

y = 10

print(x > y)
```

📤 **Output:**

```
True
```

🧠 Explanation:
Since 15 is greater than 10, the result is True.

## 🔸 Less Than (<)

Checks whether the **left value is smaller than** the right value.

💡 Example:

x = 3

y = 9

print(x < y)

📤 **Output:**

True

🧠 Explanation:
3 is smaller than 9, so it returns True.

## 🔸 Greater Than or Equal To (>=)

Checks whether the **left value is greater than or equal to** the right value.

💡 Example:

marks = 70

passing_marks = 70

print(marks >= passing_marks)

📤 **Output:**

True

🧠 Explanation:
Even though both values are equal, the condition includes equality (>=), so it returns True.

## 🔸 Less Than or Equal To (<=)

Checks whether the **left value is smaller than or equal to** the right value.

💡 Example:

x = 8

y = 10

print(x <= y)

📤 **Output:**

True

🧠 Explanation:
8 is less than 10, so condition is True.

## 🧩 Real-World Example: Age Check

```
age = int(input("Enter your age: "))


if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote yet.")
```

📤 **Output Example:**

Enter your age: 16

You are not eligible to vote yet.

🧠 Explanation:

- Python compares if age >= 18
- If True → prints “eligible”,
- If False → prints “not eligible”.

## 🧩 Example 2: Password Match

```
password = "python123"

user_input = input("Enter password: ")


if user_input == password:
    print("Access Granted ✅")
else:
    print("Access Denied ❌")
```

📤 **Output:**

Enter password: python123

Access Granted ✅

🧠 Explanation:

- == operator compares both strings.
- Only if both match, access is granted.

## 🧩 Example 3: Comparing Numbers

a = 50

b = 75

print(a == b)  # False

print(a != b)  # True

print(a > b)  # False

print(a < b)  # True

print(a >= b)  # False

print(a <= b)  # True

📤 **Output:**

False

True

False

True

False

True

## 🧠 Comparison of Strings

You can also compare **strings** alphabetically (lexicographically).

👉 Python compares strings **character by character** using Unicode values.

### 💡 Example:

print("apple" == "apple")  # True

```
print("apple" != "banana") # True

print("apple" > "banana")  # False

print("apple" < "banana")  # True
```

📤 **Output:**

```
True

True

False

True
```

🧠 Explanation:
Alphabetically, "a" comes before "b", so "apple" < "banana" is True.

## ⚡ Mixing Comparison with Variables

You can combine comparisons with variables and expressions.

### 💡 Example:

```
x = 5

y = 10

z = 15


print(x + y == z)  # True (5+10 = 15)

print(z - y != x)  # False (15-10 = 5, same as x)
```

📤 **Output:**

```
True

False
```

## 🔄 Chained Comparisons

Python allows you to **chain multiple comparisons** together.

### 💡 Example:

```
x = 10
```

```
print(5 < x < 20)  # True
```

📤 **Output:**

```
True
```

🧠 Explanation:
It checks both conditions at once:
(5 < x) and (x < 20) → both are True.

## 🧩 Example: Grade Checker

```
marks = int(input("Enter your marks: "))


if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 60:
    print("Grade: C")
else:
    print("Grade: Fail")
```

📤 **Output Example:**

```
Enter your marks: 82
Grade: B
```

🧠 Explanation:
Comparison operators check multiple conditions one by one and return results accordingly.

## 🧠 Comparison Operators with Boolean Values

```
a = True
b = False


print(a == b)  # False
```

```
print(a != b)  # True
```

📤 **Output:**

```
False

True
```

🧠 Explanation:
Python treats True as 1 and False as 0 internally.

## ⚠️ Common Mistake

🚫 **Don’t use = for comparison**
✅ Use == instead.

### ❌ Wrong:

```
if x = 5:
    print("Hello")
```

### ✅ Correct:

```
if x == 5:
    print("Hello")
```

Reason:
= is used for assignment, not comparison.

## 🧠 Quick Recap Table

| Operator | Meaning | Example | Result |
|---|---|---|---|
| == | Equal to | 5 == 5 | True |
| != | Not equal to | 5 != 3 | True |
| > | Greater than | 10 > 7 | True |
| < | Less than | 2 < 9 | True |
| >= | Greater or equal | 8 >= 8 | True |
| <= | Less or equal | 6 <= 10 | True |

## 🧩 Practice Exercises

Check if a number is greater than 100.
Compare two strings and print if they are equal.
Write a program to check if a number is positive, negative, or zero.
Compare age of two people and print who is older.
Use chained comparison to check if a number lies between 1 and 10.

## 💡 Example Challenge

```
# Program to check whether the entered number is in range 1-100


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


if 1 <= num <= 100:
    print("✅ Number is in range")
else:
    print("❌ Number is out of range")
```

📤 **Output Example:**

```
Enter a number: 45
```

✅ Number is in range

## 🧩 Summary Recap

✅ Comparison operators **compare values**
✅ They **return True or False**
✅ Used in **if-else**, **loops**, and **conditions**
✅ Work with **numbers**, **strings**, and **variables**
✅ Very useful in **decision-making logic**

Day 10: Logical Operators

# 🧠 Day 10: Logical Operators in Python

## 🔷 What are Logical Operators?

Logical Operators are used to combine multiple **conditions (Boolean expressions)** and return either **True** or **False** as a result.
They are mainly used in **decision-making** and **control flow statements** like if, while, etc.

These operators work only with **Boolean values** (True or False).

## 🔹 Python has 3 Logical Operators:

| Operator | Description | Example |
|---|---|---|
| and | Returns True if **both** conditions are True | x > 5 and y < 10 |
| or | Returns True if **at least one** condition is True | x > 5 or y < 10 |
| not | Reverses the result — returns True if condition is False | not(x > 5) |

## 🧩 Syntax of Logical Operators

```
# Using 'and'
condition1 and condition2

# Using 'or'
condition1 or condition2

# Using 'not'
not condition
```

## 💡 Examples of Logical Operators

Let's understand with simple and clear examples 👇

### ✅ Example 1: Using and Operator

```
x = 10
y = 5

# Check if both conditions are true
result = (x > 5) and (y < 10)
print(result)
```

**Output:**

```
True
```

**Explanation:**

- x > 5 → 10 > 5 → ✅ True
- y < 10 → 5 < 10 → ✅ True
- Both are True → True and True = True

### ✅ Example 2: Using or Operator

```
x = 3
y = 8
```

```
result = (x > 5) or (y < 10)

print(result)
```

**Output:**

```
True
```

**Explanation:**

- x > 5 → ❌ False
- y < 10 → ✅ True
- One condition is True → False or True = True

### ✅ Example 3: Using not Operator

```
x = 10


result = not(x > 5)

print(result)
```

**Output:**

```
False
```

**Explanation:**

- x > 5 → ✅ True
- not(True) → ❌ False
  So the result is **False**.

## 🔁 Logical Operators with Real-Life Example

Imagine you are writing code for **login validation**:

```
username = "naveen"

password = "12345"


if username == "naveen" and password == "12345":

    print("✅ Login Successful!")
```

```
else:
    print("❌ Invalid credentials!")
```

**Output:**

✅ Login Successful!

**Explanation:**

- Both username and password match, so True and True = True.

## 💻 Logical Operators with Boolean Values

You can also directly use Boolean values:

```
a = True
b = False

print(a and b)  # False
print(a or b)   # True
print(not b)    # True
```

**Output:**

```
False
True
True
```

## 🧠 Truth Table Summary

| A | B | A and B | A or B | not A |
|---|---|---|---|---|
| True | True | True | True | False |
| True | False | False | True | False |
| False | True | False | True | True |
| False | False | False | False | True |

## ⚙️ Real-World Use Cases

| Use Case | Description | Example |
|---|---|---|
| ✅ Login Validation | Check if both username and password match | if user == "admin" and pass == "1234" |
| 🚦 Driving Eligibility | Check multiple conditions together | if age >= 18 and has_license == True |
| 📱 App Feature Access | Give access if at least one permission is allowed | if camera or mic: |
| 🔒 Security Check | Deny access if any rule fails | if not is_verified: |

## 🧮 Combined Example (Multiple Logical Operators)

age = 25

has_license = True

has_helmet = True

if (age >= 18) and has_license and has_helmet:

    print("You can drive safely!")

else:

    print("Driving not allowed!")

**Output:**

You can drive safely!

## ⚡ Short-Circuit Behavior

Python uses **short-circuiting** in logical operations:

- For and: If the first condition is False, Python doesn't check the second.

- For or: If the first condition is True, Python doesn't check the second.

Example:

```
def check():
    print("Checked!")
    return True

print(False and check())  # Output: False (check() not called)
print(True or check())    # Output: True (check() not called)
```

## 🎯 1 Summary

| Operator | Meaning | Returns True When | Example |
|---|---|---|---|
| and | Logical AND | Both conditions are True | x > 5 and y < 10 |
| or | Logical OR | At least one condition is True | x > 5 or y < 10 |
| not | Logical NOT | Condition is False | not(x > 5) |

## 🧩 Mini Practice Questions

1. Write a program to check if a number is between 10 and 50.
2. Write a program to check if a person is eligible to vote **and** has a valid ID.
3. Write a program to print "Access Granted" if username is correct **or** password is correct.
4. Write a program using not to deny access if a person is not verified.

# Day 11: Identity and Membership Operators

# 🧠 Day 11: Identity and Membership Operators in Python

## 🔷 What are Identity and Membership Operators?

Python provides **two special categories of operators** that make our comparisons and checks easier:

- **Identity Operators** → used to **compare memory locations** (whether two variables refer to the same object).
- **Membership Operators** → used to **check if a value is present** in a sequence (like list, string, tuple, etc).

These operators return a **Boolean value** — True or False.

## 🧩 Identity Operators

Identity operators are used to **compare the memory address** (not the value) of two objects.

| Operator | Description | Example |
|---|---|---|
| is | Returns True if both variables point to the **same object** in memory | x is y |
| is not | Returns True if both variables point to **different objects** | x is not y |

## 💻 Identity Operators – Syntax and Example

✅ Example 1: Using is

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

y = x   # both refer to the same object

z = [1, 2, 3]  # different object with same values


print(x is y)
```

print(x is z)

**Output:**

True

False

**Explanation:**

- x is y → ✅ True (same memory address)
- x is z → ❌ False (different objects, even though values are same)

## ✅ Example 2: Using is not

a = [10, 20]

b = [10, 20]

print(a is not b)

**Output:**

True

**Explanation:**

- a and b have same content, but are stored in **different memory locations**.

## 🧠 Checking Memory Address

We can verify memory addresses using id() function.

x = [1, 2, 3]

y = x

z = [1, 2, 3]

print(id(x))

print(id(y))

print(id(z))

**Output (example):**

140219320723456

140219320723456

140219320723520

You'll notice x and y have the **same id**, but z is different.

## 🧩 Membership Operators

Membership operators are used to **check if a value exists inside a sequence** like list, string, tuple, dictionary, etc.

| Operator | Description | Example |
|---|---|---|
| in | Returns True if value is **found** in the sequence | 'a' in 'apple' |
| not in | Returns True if value is **not found** in the sequence | 'z' not in 'apple' |

## 💻 Membership Operators – Syntax and Example

### ✅ Example 1: Using in

```
fruits = ['apple', 'banana', 'cherry']


print('apple' in fruits)

print('mango' in fruits)
```

**Output:**

```
True

False
```

**Explanation:**

- 'apple' is in the list → ✅ True
- 'mango' is not in the list → ❌ False

### ✅ Example 2: Using not in

```
text = "Python programming"


print('Java' not in text)

print('Python' not in text)
```

**Output:**

```
True

False
```

**Explanation:**

- 'Java' is **not** found → ✅ True
- 'Python' **is** found → ❌ False

## 🧩 Membership with Dictionary

When you use in with a dictionary, it checks only **keys**, not values.

```
person = {'name': 'Naveen', 'age': 22, 'city': 'Vijayawada'}


print('name' in person)   # True (key exists)

print('Naveen' in person) # False (value, not key)
```

**Output:**

```
True

False
```

## 💡 Combining Identity and Membership Operators

You can use both together for advanced conditions.

### Example:

```
list1 = [1, 2, 3, 4, 5]

list2 = list1

num = 3
```

```
if (num in list1) and (list1 is list2):
    print("Number found and lists are identical!")
else:
    print("Condition not met.")
```

**Output:**

Number found and lists are identical!

## 🧠 Real-World Use Cases

| Use Case | Description | Example |
|---|---|---|
| ✅ **User Authentication** | Check if username is in database | 'naveen' in users |
| 📑 **Word Search in Text** | Check if a keyword exists in a paragraph | 'python' in content |
| 🧩 **Memory Optimization** | Compare if two variables point to same object | cache_data is previous_data |
| 🔍 **Data Validation** | Ensure input is in valid list | if country in allowed_countries: |

## ⚙️ Difference Between “is” and “==”

| Operator | Checks | Example | Output |
|---|---|---|---|
| == | Value equality | [1,2,3] == [1,2,3] | ✅ True |
| is | Identity (memory) equality | [1,2,3] is [1,2,3] | ❌ False |

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

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

print(a is b)  # False → memory locations differ

## 🧮 1 Summary

| Operator | Type | Meaning | Returns True When | Example |
|---|---|---|---|---|
| is | Identity | Both objects are same | Same memory address | x is y |
| is not | Identity | Objects are different | Different memory | x is not y |
| in | Membership | Value present in sequence | Value exists | 'a' in 'apple' |
| not in | Membership | Value not present | Value missing | 'z' not in 'apple' |

## 🧩 Mini Practice Questions

1. Write a program to check if two lists refer to the same memory location.
2. Write a program to check if "Python" exists in a given string.
3. Write a program to verify if an entered username exists in the user list.
4. Write a program to check if two sets are identical using is.
5. Create a dictionary and check if a key 'id' exists in it.

## 🎯 In Simple Terms:

- **is / is not** → compare **memory identity**.
- **in / not in** → check **membership (presence of value)**.

Day 12: Bitwise Operators

# 🧠 Day 12: Bitwise Operators in Python

## 🔷 What are Bitwise Operators?

Bitwise operators in Python are used to **perform operations on individual bits** of numbers (i.e., 0s and 1s).
They are mostly used in **low-level programming**, **data compression**, **encryption**, and **hardware control**.

Every integer in Python is stored in **binary form**, and bitwise operators help us manipulate those bits directly.

## 🧩 Why Learn Bitwise Operators?

Because computers understand **binary (0s and 1s)**.
Using bitwise operators allows you to:

- Perform faster calculations.
- Optimize storage.
- Work at the hardware or network level (e.g., IP masking).

## 🔹 List of Bitwise Operators

| Operator | Name | Description | Example |
| --- | --- | --- | --- |
| & | AND | Sets each bit to 1 if both bits are 1 | a & b |
| ` | ` | OR | Sets each bit to 1 if one of two bits is 1 |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 | a ^ b |
| ~ | NOT | Inverts all bits (changes 1 → 0 and 0 → 1) | ~a |
| << | Left Shift | Shifts bits to the left by given number | a << 1 |
| >> | Right Shift | Shifts bits to the right by given number | a >> 1 |

## 💡 Understanding Binary Representation

Before we go deeper, understand how Python represents numbers in binary.

| Decimal | Binary |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |

You can use Python's built-in bin() function to see binary representation.

```
print(bin(10))  # Output: 0b1010
```

## 💻 Bitwise AND (&)

**Rule:** Returns 1 only if **both bits are 1**.

| a | b | a & b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

✅ Example:

```
a = 10  # (binary 1010)

b = 4   # (binary 0100)

print(a & b)
```

**Calculation:**

```
  1010

&  0100

=  0000  (binary) → 0 (decimal)
```

**Output:**

```
0
```

## 💻 Bitwise OR (|)

**Rule:** Returns 1 if **either bit is 1**.

| a | b | a \| b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

✅ Example:

```
a = 10  # 1010

b = 4   # 0100

print(a | b)
```

**Calculation:**

```
  1010

|  0100

=  1110  (binary) → 14 (decimal)
```

**Output:**

```
14
```

## 💻 Bitwise XOR (^)

**Rule:** Returns 1 if **only one** of the bits is 1 (not both).

| a | b | a ^ b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

✅ Example:

a = 10  # 1010

b = 4   # 0100

print(a ^ b)

**Calculation:**

  1010

^  0100

=  1110  → 14

**Output:**

14

## 💻 Bitwise NOT (~)

**Rule:** Flips all bits.
 ~a = -(a + 1)

✅ Example:

a = 10

print(~a)

**Calculation:**

~10 = -(10 + 1) = -11

**Output:**

-11

**Explanation:**
Python represents numbers using **2's complement** for negatives, hence the result is -(a+1).

## 💻 Bitwise Left Shift (<<)

**Rule:** Shifts bits to the **left** by the specified number of positions.
Each left shift multiplies the number by **2**.

✅ Example:

a = 5  # binary: 0101

print(a << 1)

print(a << 2)

**Calculation:**

a << 1 → 1010 (binary) → 10

a << 2 → 10100 (binary) → 20

**Output:**

10

20

## 💻 🔟 Bitwise Right Shift (>>)

**Rule:** Shifts bits to the **right** by the specified number of positions.
Each right shift divides the number by **2**.

✅ Example:

a = 10  # binary: 1010

print(a >> 1)

print(a >> 2)

**Calculation:**

a >> 1 → 0101 → 5

a >> 2 → 0010 → 2

**Output:**

5

2

## 🧠 1 Real-World Examples

| Use Case | Description | Example |
|---|---|---|
| 🧮 **Fast Multiplication** | Use left shift to multiply by 2 | x << 1 |
| ➗ **Fast Division** | Use right shift to divide by 2 | x >> 1 |
| 🔐 **Encryption / XOR logic** | Flip bits for encryption | encrypted = data ^ key |
| ⚙️ **Hardware Control** | Turn bits ON/OFF for specific hardware | port_value & mask |

## 💡 1 Example: Swapping Two Numbers Using XOR

a = 5

b = 7

a = a ^ b

b = a ^ b

a = a ^ b

print("a =", a)

print("b =", b)

**Output:**

a = 7

b = 5

**Explanation:**
Bitwise XOR lets us swap numbers **without using a third variable**.

## 🧮 1 Summary Table

| Operator | Name | Example | Result | Meaning |
|---|---|---|---|---|
| & | AND | 5 & 3 | 1 | Both bits 1 |
| ` | ` | OR | `5 | 3` |
| ^ | XOR | 5 ^ 3 | 6 | Only one bit 1 |
| ~ | NOT | ~5 | -6 | Invert bits |
| << | Left Shift | 5 << 1 | 10 | Multiply by 2 |
| >> | Right Shift | 5 >> 1 | 2 | Divide by 2 |

## 🧩 1 Practice Questions

1. Write a program to find the AND, OR, and XOR of two numbers.
2. Write a program to multiply a number by 8 using bitwise shift.
3. Write a program to check if a number is even or odd using bitwise operator (a & 1).
4. Swap two numbers without using a third variable.
5. Convert a decimal number to binary and perform left shift by 2.

## 🎯 1 Summary (In Simple Words)

- **& (AND):** Both bits 1 → 1
- **| (OR):** Any bit 1 → 1
- **^ (XOR):** One bit 1 → 1
- **~ (NOT):** Flip bits
- **<<:** Move bits left → multiply by 2
- **>>:** Move bits right → divide by 2

# 🧠 Day 13: If Statement in Python

## 🔷 What is an If Statement?

The **if statement** is used in Python to make **decisions** based on conditions.
It allows your program to **execute a block of code only when a condition is true** — otherwise, it skips that block.

You can think of it like a **real-life decision**:

💬 *"If it's raining, then take an umbrella."*

If it's not raining, you do nothing — that's exactly how the if statement works!

## 🔹 Syntax of if Statement

```
if condition:
    # block of code to execute when condition is True
```

✅ **Rules:**

- The if keyword starts the condition.
- The **condition** must return either True or False.
- The **colon (:)** marks the start of the block.
- The **code inside** the if block must be **indented** (4 spaces by convention).

## 💡 Simple Example

```
x = 10

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

**Output:**

```
x is greater than 5
```

**Explanation:**

- Condition: x > 5 → ✅ True

- Since the condition is true, the code inside if block runs.

## 🚫 When Condition is False

If the condition is False, Python simply **skips** the code block.

```
x = 3


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


print("Program End")
```

**Output:**

```
Program End
```

**Explanation:**

- Condition: x > 5 → ❌ False
- So, the print statement inside if does **not** execute.

## 🧩 Real-Life Example

```
temperature = 35


if temperature > 30:
    print("It's a hot day! Stay hydrated.")
```

**Output:**

```
It's a hot day! Stay hydrated.
```

**Explanation:**
The program checks if the temperature is above 30. Since it is, the message is displayed.

## 🧠 Understanding the Flow

Here's how Python reads it:

Step 1 → Check condition.

Step 2 → If True → execute the code inside `if`.

Step 3 → If False → skip the block.

Step 4 → Continue with the next line.

## 💻 Example with Variables and Expressions

```
marks = 90


if marks >= 75:
    print("Excellent! You got distinction.")
```

**Output:**

```
Excellent! You got distinction.
```

## ⚙️ Using Logical Operators with if

You can combine multiple conditions using and, or, and not.

```
age = 20
has_license = True


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

**Output:**

```
You are eligible to drive.
```

## 🧮 Checking Even or Odd Number

```
number = 7


if number % 2 == 0:
    print("Even number")
```

**Output:**

(no output)

**Explanation:**
7 % 2 == 0 → False
So nothing is printed.

Try with number = 8 — it will print Even number.

## 🎯 1 Nested If Statements

You can also **put an if inside another if** (called *nested if*).

```
age = 25

citizenship = "Indian"


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

**Output:**

You are eligible to vote in India.

**Explanation:**
Both conditions are True, so the inner message is printed.

## 🧩 1 If Statement with Input

You can take user input to check conditions dynamically.

```
age = int(input("Enter your age: "))


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

**Example Run:**

Enter your age: 20

You are eligible to vote.

## 🔁 1 Indentation is Important!

Python **does not use brackets {}** like other languages (C, Java, etc.).
Instead, it relies on **indentation** (spaces or tabs).

✅ Correct:

```
if True:
    print("Hello")
```

❌ Incorrect (will cause error):

```
if True:
print("Hello")
```

**Output:**

IndentationError: expected an indented block

## 💡 1 Multiple If Statements (Independent Conditions)

Each if is checked separately.

```
x = 15

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

if x % 5 == 0:
    print("x is divisible by 5")

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

**Output:**

x is greater than 10

x is divisible by 5

x is less than 20

## 🧠 1 Real-World Scenarios for if

| Scenario | Condition | Example |
|---|---|---|
| ✅ Voting Eligibility | Check if user is 18 or older | if age >= 18: |
| 💰 Bank Balance | Check if balance > 0 before withdrawal | if balance > 0: |
| 📱 App Login | Check if username and password are correct | if user == "admin": |
| 🚦 Traffic Control | Check if light is red | if light == "red": |

## 🧩 1 Combined Example

```
marks = int(input("Enter your marks: "))


if marks >= 90:
    print("Grade: A+")
if marks >= 75:
    print("You passed with distinction.")
if marks < 40:
    print("You failed the exam.")
```

**Input/Output Example:**

Enter your marks: 85

You passed with distinction.

## 🧮 1 Key Points to Remember

| Feature | Description |
|---|---|
| if | Used to execute code only when condition is True |
| Condition | Must return True or False |
| Indentation | Required for code block |
| Colon : | Must follow if |
| Multiple if | Checked independently |

## 🧩 1 Mini Practice Questions

1. Write a program to check if a number is positive.
2. Write a program to check if the entered number is greater than 100.
3. Write a program that checks if a person is eligible for a driving license (age ≥ 18).
4. Write a program that prints “Weekend” if the day is Saturday.
5. Write a program to print “High Score!” if score > 80.

## 🎯 1 Summary (In Simple Words)

- **If** = condition check.
- Executes **only** if the condition is **True**.
- Indentation and colon are mandatory.
- Used for **decision making** in programs.

✅ **Real-life Analogy:**

If you have money, then you can buy ice cream.
Otherwise, you can’t — that’s exactly how Python’s if works! 🍦

Day 14: If-Else Statement

# 🧠 Day 14: If-Else Statement in Python

## 🔍 1. Introduction

In programming, decisions are important.
The **if-else statement** allows your program to **choose between two paths** — one when a condition is **true** and another when it is **false**.

In simple words:

“If something is true, do this. Otherwise, do something else.”

It’s like your brain deciding:

- **If** it’s raining 🌧️ → Take an umbrella ☂️
- **Else** → Go outside freely 😎

## 🧩 2. Syntax

```
if condition:
    # code block executed when condition is True
else:
    # code block executed when condition is False
```

## 💡 3. Example 1 — Basic Example

```
age = 18

if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

### 🧠 Explanation:

- The program checks whether age >= 18.
- If **True**, it prints "You are eligible to vote."
- If **False**, it prints "You are not eligible to vote."

### ✅ Output:

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

## 💡 4. Example 2 — Real-Life Example

```
temperature = 35


if temperature > 30:
    print("It’s a hot day.")
else:
    print("It’s a cool day.")
```

### Explanation:

- If the temperature is more than 30°C, it’s hot.
- Otherwise, it’s cool.

✅ **Output:**

```
It’s a hot day.
```

## 💡 5. Example 3 — With Input from User

```
marks = int(input("Enter your marks: "))


if marks >= 50:
    print("You passed the exam! 🎉")
else:
    print("You failed the exam. 😔")
```

### Explanation:

- The user enters marks.
- If marks are **50 or above**, print pass.
- Else, print fail.

✅ **Output (if user enters 45):**

```
You failed the exam. 😔
```

## 🧮 6. Example 4 — Even or Odd Check

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

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

### Explanation:

- If remainder after dividing by 2 is 0 → number is **even**.
- Otherwise → **odd**.

✅ **Output (for 7):**

```
Odd number
```

## 🔁 7. Flow of Execution

| Step | What Happens |
|---|---|
| | Python checks the condition in the if statement |
| | If condition is True → Executes if block |
| | If condition is False → Executes else block |
| | Program continues after the if-else structure |

## 🧠 8. Example 5 — Salary Bonus Logic

```
salary = 50000
experience = 3

if experience >= 5:
    bonus = salary * 0.1
else:
```

```
    bonus = salary * 0.05


print("Bonus:", bonus)
```

### Explanation:

- Employees with ≥ 5 years experience get 10% bonus.
- Others get 5%.

✅ **Output:**

```
Bonus: 2500.0
```

## ⚙️ 9. Nested If-Else (Mini Example)

You can also **nest** if-else statements inside each other.

```
num = 0


if num > 0:
    print("Positive")
else:
    if num < 0:
        print("Negative")
    else:
        print("Zero")
```

✅ **Output:**

```
Zero
```

## 🧩 10. Real-World Analogy

Think of **if-else** like a **traffic light system** 🚦

- **If** light is green → Move 🟢
- **Else** (red/yellow) → Stop 🔴

Python uses this logic to decide what action to take based on given conditions.

## ⚠️ 11. Common Mistakes

❌ Missing indentation:

```
if True:
print("Hello")   # ❌ Wrong
```

✅ Correct:

```
if True:
    print("Hello")
```

❌ Using = instead of == for comparison:

```
if x = 10:  # ❌ Wrong
```

✅ Correct:

```
if x == 10:  # ✅ Right
```

## 🧾 12. Key Points to Remember

✔️ if → checks condition
✔️ else → executes when if condition is false
✔️ Only **one** else per if statement
✔️ Indentation matters (use 4 spaces or 1 tab)
✔️ Conditions should return **True** or **False**

## 🎯 13. Practice Questions

1. Write a Python program to check if a number is positive or negative.
2. Take a user's age as input and print whether they are a child or adult.
3. Write a program to check if the entered year is leap year or not.
4. Write a program to compare two numbers and print which one is greater.
5. Write a program that checks if a person is eligible for a driving license (age ≥ 18).

## 🏁 14. Summary

| Keyword | Purpose |
| --- | --- |
| if | Checks a condition |
| else | Executes alternative code if condition is false |
| : | Indicates start of a code block |
| **Indentation** | Defines which code belongs to if or else |

Day 15: Elif Statement

# 🧠 Day 15: Elif Statement in Python

## 🔍 1. Introduction

In programming, sometimes you need to **check multiple conditions**, not just one or two.

That's where the **elif (short for "else if")** statement comes in!

The **elif statement** allows your program to test **many conditions** and execute the code block for the **first condition that's True**.

Think of it like this 👇

"If not this, maybe that. If not that, maybe the next one!"

## 🎯 2. Real-Life Analogy

Imagine your daily routine 🌞

- **If** it's Monday → Go to work
- **Elif** it's Saturday → Go for shopping
- **Elif** it's Sunday → Take rest
- **Else** → Invalid day

Python behaves the same way!

## 🧩 3. Syntax

```
if condition1:
    # Executes if condition1 is True
elif condition2:
    # Executes if condition1 is False and condition2 is True
elif condition3:
    # Executes if above conditions are False and this is True
else:
    # Executes if all above conditions are False
```

## 💡 4. Example 1 — Grading System

```
marks = 85

if marks >= 90:
    print("Grade: A")
elif marks >= 80:
    print("Grade: B")
elif marks >= 70:
    print("Grade: C")
elif marks >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

### 🧠 Explanation:

- The program checks each condition **in order**.
- As soon as one condition is True, Python executes it and **skips the rest**.

### ✅ Output:

Grade: B

## 💡 5. Example 2 — Temperature Checker

```
temp = 20


if temp > 30:
    print("It's Hot 🥵")
elif temp > 20:
    print("It's Warm 🌤️")
elif temp > 10:
    print("It's Cool 🌥️")
else:
    print("It's Cold ❄️")
```

✅ **Output:**

It's Cool 🌥️

### Explanation:

- temp = 20 → first condition (temp > 30) is False
- second (temp > 20) is False
- third (temp > 10) is True → prints “It’s Cool”

## 💡 6. Example 3 — Traffic Light System 🚦

```
light = "green"


if light == "red":
    print("Stop 🚫")
elif light == "yellow":
    print("Get Ready ⚠️")
elif light == "green":
```

```
    print("Go ✅")
else:
    print("Invalid Signal")
```

✅ **Output:**

```
Go ✅
```

## 💡 7. Example 4 — Login System

```
username = "Naveen"
password = "1234"

if username == "Admin":
    print("Welcome Admin!")
elif username == "Naveen" and password == "1234":
    print("Welcome Naveen!")
else:
    print("Invalid Credentials ❌")
```

✅ **Output:**

```
Welcome Naveen!
```

## 🧮 8. Flow of Execution

| Step | Description |
| --- | --- |
| | Python checks the first if condition |
| | If True → executes that block and skips the rest |
| | If False → checks the next elif |
| | If none match → executes the else block |

## ⚙️ 9. Example 5 — Movie Ticket Pricing 🎟️

```
age = 10


if age < 5:
    print("Free Ticket")
elif age <= 12:
    print("Child Ticket - ₹100")
elif age <= 18:
    print("Teen Ticket - ₹150")
elif age <= 60:
    print("Adult Ticket - ₹200")
else:
    print("Senior Citizen Ticket - ₹120")
```

✅ **Output:**

```
Child Ticket - ₹100
```

## 🧠 10. Key Concept

- You can use **multiple elifs** in one program.
- Only **one** block executes (the first True one).
- The else part is **optional**.
- If none of the conditions are True and there's no else, **nothing happens**.

## 🧩 11. Nested Elif vs. Nested If

Instead of writing nested if-else, you can simplify your code with elif.

❌ **Nested If (complex):**

```
if x > 0:
    print("Positive")
else:
    if x == 0:
        print("Zero")
    else:
        print("Negative")
```

✅ **Using Elif (clean):**

```
if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

## ⚠️ 12. Common Mistakes

❌ Using multiple if instead of elif:

```
if x > 0:
    print("Positive")
if x == 0:
```

```
    print("Zero")
else:
    print("Negative")
```

👉 Both if statements can execute — which is not intended!

✅ Correct:

```
if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

## 🗒️ 13. Key Points to Remember

| Keyword | Meaning |
|---|---|
| if | First condition check |
| elif | Used for multiple conditional checks |
| else | Executes if all previous conditions fail |
| : | Starts a new block |
| **Order matters** | Python executes top to bottom |

## 🎯 14. Practice Questions

1. Write a Python program to determine the grade based on marks:
   - 90+ → A
   - 80–89 → B
   - 70–79 → C
   - else → Fail
2. Take a number as input and print whether it's positive, negative, or zero.
3. Write a Python program to display day name based on a number (1 = Monday ... 7 = Sunday).
4. Create a program that tells which season it is based on the month number.
5. Take age as input and classify:
   - Below 13 → Child
   - 13–19 → Teen
   - 20–59 → Adult
   - 60+ → Senior Citizen

## 🏁 15. Summary

| Concept | Description |
|---|---|
| if | Checks the first condition |
| elif | Checks next conditions (in order) |
| else | Executes when all fail |
| **Execution** | Stops at the first True condition |
| **Usage** | Best for multiple decision paths |

✅ **In short:**

elif is your way to handle **multiple conditions** neatly in Python — instead of messy nested if-else structures.

# 🧠 Day 16: Nested If Statements in Python

## 🔍 1. Introduction

Sometimes, you need to check **one condition inside another**.
This is called a **Nested If Statement** — one if inside another if.

In simple terms:

"If this is true, then check another condition."

It's like real life:

If it's your **birthday**, then check **if you have friends coming** — if yes, **celebrate!** 🎉

## 🎯 2. What is a Nested If?

A **Nested If** means you put one if (or even elif / else) statement inside another.
It allows your program to make **decisions within decisions**.

## 🧩 3. Syntax

```
if condition1:
    # Outer if block
    if condition2:
        # Inner if block
        # Executes when both condition1 and condition2 are True
    else:
        # Inner else block
else:
    # Outer else block
```

## 💡 4. Example 1 — Eligibility Check

```
age = 20
```

```
citizen = "Indian"


if age >= 18:
    if citizen == "Indian":
        print("You are eligible to vote in India.")
    else:
        print("You are not an Indian citizen.")
else:
    print("You are not old enough to vote.")
```

✅ **Output:**

```
You are eligible to vote in India.
```

🧠 **Explanation:**

- Outer if: checks if age ≥ 18
- Inner if: checks if citizenship is Indian
- Both must be true to print the first line

## 💡 5. Example 2 — Bank Loan Approval System

```
income = 50000
credit_score = 750


if income >= 40000:
    if credit_score >= 700:
        print("Loan Approved ✅")
    else:
        print("Loan Denied ❌ — Low credit score")
else:
    print("Loan Denied ❌ — Low income")
```

✅ **Output:**

Loan Approved ✅

## 💡 6. Example 3 — Student Result System

```
marks = 85


if marks >= 40:
    if marks >= 75:
        print("Distinction 🏅")
    elif marks >= 60:
        print("First Class 🎓")
    elif marks >= 40:
        print("Pass 👍")
else:
    print("Fail ❌")
```

✅ **Output:**

Distinction 🏅

### 🧠 Explanation:

- Outer if ensures student passed (marks ≥ 40)
- Inside that, another set of checks decides the class level

## 💡 7. Example 4 — Login System

```
username = "Naveen"
password = "1234"


if username == "Naveen":
    if password == "1234":
        print("Login Successful 🎉")
    else:
```

```
    print("Incorrect Password ❌")
else:
  print("Invalid Username ❌")
```

✅ **Output:**

Login Successful 🎉

### Explanation:

- First check username
- Then check password **only if username is correct**

## 🧮 8. Example 5 — Number Type Checker

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


if num >= 0:
  if num == 0:
    print("Number is Zero")
  else:
    print("Number is Positive")
else:
  print("Number is Negative")
```

✅ **Output (if user enters -8):**

Number is Negative

## 🔁 9. Flow of Execution

| Step | Description |
| --- | --- |
| | Python checks the **outer if** condition |
| | If True → executes the inner if block |
| | If False → skips the inner block entirely |
| | The outer else executes if the first condition fails |

## 🧠 10. Real-Life Analogy

Imagine you are at an amusement park 🎡

- **If** you have a ticket 🎟️
  - **If** you are above 12 years → allowed on roller coaster 🎢
  - **Else** → only allowed on merry-go-round 🎠
- **Else** → no entry ❌

That's **nested if** logic!

## ⚙️ 11. Example 6 — Exam Result (Advanced)

```
marks = int(input("Enter your marks: "))


if marks >= 40:
    print("You Passed! 🎉")
    if marks >= 90:
        print("Excellent! Grade: A+ 🏆")
    elif marks >= 75:
        print("Good Job! Grade: A")
    elif marks >= 60:
        print("Well Done! Grade: B")
    else:
```

```
        print("Keep Improving! Grade: C")
else:
    print("You Failed. Try Again! ❌")
```

✅ **Output (if user enters 88):**

```
You Passed! 🎉
Good Job! Grade: A
```

## ⚠️ 12. Common Mistakes

❌ **Wrong indentation:**

```
if x > 0:
print("Positive")
    if x < 100:
        print("Less than 100")
```

✅ **Correct indentation:**

```
if x > 0:
    print("Positive")
    if x < 100:
        print("Less than 100")
```

❌ **Missing outer condition:**
You can't use an inner if without a properly nested structure.

## 🧾 13. Key Points to Remember

| Concept | Description |
| --- | --- |
| if inside if | Called Nested If |
| else inside if | Works normally inside |
| Indentation | Very important (defines levels) |
| Execution | Inner if runs only if outer if is True |
| Nesting Depth | Avoid nesting too deep (code becomes complex) |

## 🎯 14. Practice Questions

1. Write a program to check if a person is eligible for a job:
   - Age ≥ 18
   - If yes, check qualification: Graduate or not
2. Write a program to determine if a number is **positive**, **negative**, or **zero** using nested if.
3. Write a program for **restaurant discount**:
   - If bill > 1000 → check if member:
     - If yes → 20% discount
     - Else → 10% discount
   - Else → no discount
4. Write a program to verify if a student passes:
   - Marks ≥ 40 → pass
   - Inside that, check if marks ≥ 75 → distinction
5. Write a program to check eligibility for an exam:
   - If attendance ≥ 75%
     - If marks ≥ 50 → allowed to sit
     - Else → not allowed

## 🏁 15. Summary

| Keyword | Meaning |
| --- | --- |
| Nested If | Decision inside another decision |
| Outer If | Main condition |
| Inner If | Secondary condition inside main one |
| Indentation | Defines nesting levels |
| Else | Executes if the condition at the same level is False |

✅ **In short:**

A **Nested If** statement lets you make decisions inside decisions —
like a flow of questions that narrows down your logic step by step.

Day 17: While Loop

# 🧠 Day 17: While Loop in Python

## 🔍 1. Introduction

Imagine you are doing something **again and again** until a certain condition becomes false.
That's exactly what a **while loop** does in Python.

"Repeat a block of code **while** a condition is true."

For example 👇

While there is **water in the bottle**, keep drinking.
Once it's **empty**, stop.

So, a **while loop** keeps running until the condition becomes **False**.

## 🧩 2. Syntax

```
while condition:
    # Code block to execute repeatedly
```

### 🧠 Explanation:

- Python first checks the **condition**.
- If it's **True**, it executes the code block.
- After each iteration, it **checks the condition again**.
- Loop stops when the condition becomes **False**.

## 💡 3. Example 1 — Basic Example

```
count = 1

while count <= 5:
    print("Hello Python!")
    count += 1
```

### ✅ Output:

```
Hello Python!
Hello Python!
Hello Python!
Hello Python!
Hello Python!
```

### Explanation:

- count starts at 1
- The loop runs **while count ≤ 5**
- Each time it prints and increments count by 1
- When count becomes 6, condition is False → loop stops

## 💡 4. Example 2 — Counting Numbers

```
i = 1

while i <= 10:
```

```
    print(i)
    i += 1
```

✅ **Output:**

```
1
2
3
4
5
6
7
8
9
10
```

**Explanation:**

- Starts at 1, runs until 10
- Each time increases i by 1

## 💡 5. Example 3 — Sum of First 5 Numbers

```
num = 1
total = 0

while num <= 5:
    total += num
    num += 1

print("Sum =", total)
```

✅ **Output:**

Sum = 15

**Explanation:**

- Adds numbers 1 + 2 + 3 + 4 + 5
- The loop stops when num becomes 6

## 💡 6. Example 4 — Print Even Numbers

```
num = 2


while num <= 10:
    print(num)
    num += 2
```

✅ **Output:**

2

4

6

8

10

## 💡 7. Example 5 — Countdown Timer

```
count = 5


while count > 0:
    print(count)
    count -= 1


print("Time's Up! ⏰")
```

✅ **Output:**

5

4

3

2

1

Time's Up! ⏰

## 🎯 8. Real-Life Analogy

Imagine a washing machine 🧺

- It keeps rotating **while time > 0**
- When time becomes 0 → it stops automatically

That's exactly how **while loop** works —
It keeps going **while** a condition remains true.

## 🔁 9. Flow of Execution

| Step | Description |
|---|---|
| | Python checks the condition |
| | If condition is True → executes the block |
| | After executing, goes back to check condition again |
| | Repeats until condition becomes False |
| | When False → loop ends and control moves below the loop |

## 💣 10. Infinite Loop

If the condition **never becomes False**, the loop will run forever.

⚠️ **Example (Infinite Loop):**

```
x = 1
```

```
while x <= 5:
    print("Hello")
```

👉 Since x never changes, the condition x <= 5 stays True forever.

✅ **Fix:**

```
x = 1
while x <= 5:
    print("Hello")
    x += 1
```

## ⚙️ 11. Example 6 — Password Checking System

```
password = ""

while password != "python123":
    password = input("Enter your password: ")

print("Access Granted ✅")
```

✅ **Output:**

```
Enter your password: hello
Enter your password: 123
Enter your password: python123
Access Granted ✅
```

### Explanation:

- Keeps asking for input until correct password is entered

## 💡 12. Example 7 — Multiplication Table

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

```
while i <= 10:
    print(f"{n} x {i} = {n*i}")
    i += 1
```

✅ **Output (for n = 5):**

5 x 1 = 5

5 x 2 = 10

...

5 x 10 = 50

## 🧠 13. Nested While Loop

You can also use a **while loop inside another while loop**.

Example:

```
i = 1

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

✅ **Output:**

i = 1, j = 1

i = 1, j = 2

i = 2, j = 1

i = 2, j = 2

i = 3, j = 1

i = 3, j = 2

## ⚠️ 14. Common Mistakes

❌ **Forgetting to update the variable**
→ Causes infinite loop.

❌ **Wrong condition**
→ Loop may not run even once.

❌ **Incorrect indentation**
→ Causes syntax errors.

## 🧾 15. Key Points to Remember

| Concept | Description |
|---|---|
| while | Keyword for looping while condition is True |
| Condition | Must become False eventually |
| += | Commonly used to increment/decrement variable |
| Infinite Loop | Happens if condition never becomes False |
| Can combine with else | Executes when loop finishes normally |

## 💡 16. Example 8 — While with Else

```
x = 1


while x <= 3:
    print(x)
    x += 1
else:
    print("Loop finished successfully ✅")
```

✅ **Output:**

1

2

3

Loop finished successfully ✅

### Explanation:

- The else part executes only when the loop completes normally (not broken by break).

## 🎯 17. Practice Questions

1. Print numbers from 1 to 10 using a while loop.
2. Find the sum of first 10 natural numbers.
3. Write a program to print even numbers from 1 to 20.
4. Create a countdown timer from 10 to 1.
5. Take a number as input and print its multiplication table.
6. Keep asking the user for a password until they enter the correct one.
7. Use a nested while loop to print a pattern of stars ⭐.
8. Write a while loop that prints all numbers divisible by 5 between 1 and 50.
9. Find factorial of a number using while loop.
10. Write a while loop that keeps asking numbers until the user enters 0.

## 🏁 18. Summary

| Concept | Description |
|---|---|
| While Loop | Repeats code while condition is True |
| Condition | Checked before every iteration |
| Break | Can stop the loop manually |
| Else | Runs after loop finishes normally |
| Infinite Loop | Avoid by updating variables correctly |

✅ **In short:**

The **while loop** is like a “repeat until done” structure —
Python keeps running your code **as long as** the condition remains True.

Day 18: For Loop

# 🧠 Day 18: For Loop in Python

## 🔍 1. Introduction

The **for loop** in Python is one of the most used loops.
It allows you to **iterate (go through)** a sequence (like a list, string, tuple, dictionary, or range) and perform actions for **each item** in that sequence.

Think of it like this 👇

“For each student in the classroom, take attendance.”
“For each fruit in the basket, print its name.”

That’s exactly what Python’s for loop does — it **repeats a block of code** for every element in a collection.

## 🧩 2. Syntax

```
for variable in sequence:

    # code block to execute
```

### Explanation:

- **variable** → gets each value from the sequence one by one.
- **sequence** → could be a list, tuple, string, range, etc.
- **code block** → runs for each value.

## 💡 3. Example 1 — Looping Over a List

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



for fruit in fruits:

    print(fruit)
```

### Explanation:

- fruit takes each value from the list — one at a time.
- The loop runs **3 times** (for apple, banana, mango).

✅ **Output:**

apple

banana

mango

## 💡 4. Example 2 — Looping Through a String

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

✅ **Output:**

P

y

t

h

o

n

### Explanation:

The loop iterates over each character in the string "Python".

## 💡 5. Example 3 — Using Range() Function

range() generates a sequence of numbers.

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

✅ **Output:**

0

1

2

3

4

**Explanation:**

- range(5) → gives numbers from **0 to 4**.
- Loop runs **5 times**.

## ⚙️ 6. Example 4 — Range with Start and End

```
for i in range(2, 6):
    print(i)
```

✅ **Output:**

2

3

4

5

**Explanation:**

- range(2, 6) starts from **2** and ends at **5** (stop value is excluded).

## 💡 7. Example 5 — Range with Step Value

```
for i in range(0, 10, 2):
    print(i)
```

✅ **Output:**

0

2

4

6

8

**Explanation:**

- range(start, stop, step) → steps by 2 each time.

## 🧮 8. Example 6 — Sum of First N Numbers

```
n = 5

total = 0


for i in range(1, n + 1):
    total += i


print("Sum:", total)
```

✅ **Output:**

```
Sum: 15
```

### Explanation:

Adds 1 + 2 + 3 + 4 + 5 using a loop.

## 🧠 9. Example 7 — Loop with If Condition

```
for num in range(1, 10):
    if num % 2 == 0:
        print(num, "is even")
    else:
        print(num, "is odd")
```

✅ **Output:**

```
1 is odd

2 is even

3 is odd

...

9 is odd
```

**Explanation:**

- Loop iterates numbers from 1 to 9.
- Checks whether each number is even or odd.

## 💻 10. Example 8 — Looping Through a List of Tuples

students = [("Naveen", 90), ("Ravi", 85), ("Kiran", 92)]

for name, marks in students:

    print(name, "scored", marks)

✅ **Output:**

Naveen scored 90

Ravi scored 85

Kiran scored 92

## 🔁 11. Example 9 — Nested For Loop

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

for i in range(1, 4):

    for j in range(1, 4):

        print(i, j)

✅ **Output:**

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

**Explanation:**

- Outer loop → runs 3 times.
- Inner loop → runs 3 times for each outer iteration.
- Total executions = 3 × 3 = 9 times.

## 📦 12. Example 10 — Using For Loop with Dictionary

student = {"name": "Naveen", "age": 21, "course": "Python"}

for key, value in student.items():

    print(key, ":", value)

✅ **Output:**

name : Naveen

age : 21

course : Python

## ⚙️ 13. Loop Control Statements

| Statement | Meaning |
|---|---|
| break | Stops the loop completely |
| continue | Skips the current iteration |
| pass | Does nothing (placeholder) |

Example — Using break and continue

for num in range(1, 6):

    if num == 3:

        continue  # skips 3

    if num == 5:

```
        break    # stops loop at 5
    print(num)
```

✅ **Output:**

1

2

4

## 🧩 14. Real-Life Example

Imagine you're sending emails to a list of users 👇

```
users = ["John", "Naveen", "Sara"]


for user in users:
    print("Email sent to:", user)
```

✅ **Output:**

Email sent to: John

Email sent to: Naveen

Email sent to: Sara

🎯 **This is how automation and batch tasks are done in real-world Python scripts.**

## ⚠️ 15. Common Mistakes

❌ Forgetting indentation:

```
for i in range(5):
print(i)   # ❌ Wrong
```

✅ Correct:

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

❌ Modifying the list while iterating.
✅ Instead, create a new list to store results.

## 🗒️ 16. Key Points to Remember

✔ for loop works with sequences (list, string, tuple, etc.)
✔ range() is commonly used for number loops
✔ The **loop variable** changes automatically in each iteration
✔ Can be nested
✔ Can use break, continue, and pass inside

## 🏁 17. Practice Questions

1. Print numbers from 1 to 10 using a for loop.
2. Print the multiplication table of a given number.
3. Print all even numbers between 1 and 50.
4. Calculate factorial of a given number using for loop.
5. Count vowels in a given string.
6. Display all elements of a list using a for loop.
7. Print sum of digits of a number using for loop.

## 🎯 18. Summary

| Concept | Description |
|---|---|
| for variable in sequence: | Basic for loop structure |
| range(start, stop, step) | Generates number sequence |
| break | Exit loop |
| continue | Skip iteration |
| nested for | Loop inside loop |

✅ **In short:**
The **for loop** helps you **repeat actions efficiently**, making Python automation, data processing, and iteration over collections super easy.

Day 19: Range Function

# 🧠 Day 19: The range() Function in Python

## 🔍 1. Introduction

The **range() function** is one of the most important tools in Python, especially when working with **loops**.

It's used to **generate a sequence of numbers** — usually for iteration in **for loops** or for performing repetitive tasks.

Think of it like this 👇

"Give me numbers starting from 1 to 10 — I'll go through each of them one by one."

That's exactly what range() does — it creates a **sequence of integers** that you can loop over.

## 🧩 2. Syntax

range(start, stop, step)

**Parameters:**

| Parameter | Description | Default |
|---|---|---|
| start | The number to start from | 0 |
| stop | The number to stop before (excluded) | — |
| step | The difference between each number | 1 |

## 💡 3. Simplest Example

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

✅ **Output:**

0

1

2

3

4

### Explanation:

- Starts from **0** by default.
- Stops **before 5** (stop value excluded).
- Increments by **1** each time.

## 💡 4. Example with Start and Stop

```
for i in range(2, 6):
    print(i)
```

✅ **Output:**

2

3

4

5

### Explanation:

- Starts from **2**.
- Stops **before 6**.
- So it prints **2, 3, 4, 5**.

## 💡 5. Example with Step Value

```
for i in range(0, 10, 2):
    print(i)
```

✅ **Output:**

0

2

4

6

8

### Explanation:

- Starts from 0, goes up to 10 (exclusive).
- Steps by **2** each time → skips odd numbers.

## 💡 6. Example with Negative Step (Reverse Counting)

```
for i in range(10, 0, -1):
    print(i)
```

✅ **Output:**

10

9

8

7

6

5

4

3

2

1

### Explanation:

- Starts at 10.
- Ends before 0.
- Decreases by 1 each loop (-1 step value).

## ⚙️ 7. Example — Print Even Numbers

```
for i in range(2, 21, 2):
    print(i)
```

✅ **Output:**

2

4

6

8

10

12

14

16

18

20

**Explanation:**

- Starts from 2, goes till 20, increases by 2 each time.

## 💻 8. Example — Print Odd Numbers

```
for i in range(1, 20, 2):
    print(i)
```

✅ **Output:**

1

3

5

7

9

11

13

15

17

19

## 🧮 9. Example — Generate a Sequence and Convert to List

```
numbers = list(range(1, 6))

print(numbers)
```

✅ **Output:**

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

### Explanation:

- list(range(1, 6)) creates a list of numbers from 1 to 5.

## 💡 10. Example — Using Range in a Calculation

Let's calculate the sum of the first 10 natural numbers 👇

```
total = 0


for i in range(1, 11):
    total += i


print("Sum:", total)
```

✅ **Output:**

```
Sum: 55
```

### Explanation:

- Loop adds numbers 1 → 10 one by one.
- range(1, 11) stops before 11, so last value = 10.

## 🧠 11. Example — Multiplication Table

```
num = 5


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

✅ **Output:**

5 x 1 = 5

5 x 2 = 10

...

5 x 10 = 50

## 🔁 12. Example — Reverse Multiplication Table

num = 3

for i in range(10, 0, -1):

  print(num, "x", i, "=", num * i)

✅ **Output:**

3 x 10 = 30

3 x 9 = 27

...

3 x 1 = 3

## 🧩 13. Example — Loop with Conditional Logic

for i in range(1, 11):

  if i % 2 == 0:

    print(i, "is even")

  else:

    print(i, "is odd")

✅ **Output:**

1 is odd

2 is even

...

10 is even

## 🔁 14. Example — Nested Range

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

✅ **Output:**

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

### Explanation:

- Outer loop runs 3 times.
- Inner loop runs 3 times per outer iteration.

## 💡 15. Example — Negative Ranges

```
for i in range(-3, 4):
    print(i)
```

✅ **Output:**

```
-3
-2
-1
0
1
```

2

3

## ⚙️ 16. Important Notes

| Behavior | Example | Output |
|---|---|---|
| Start Default = 0 | range(5) | 0 to 4 |
| Stop is Exclusive | range(1, 5) | 1, 2, 3, 4 |
| Step Default = 1 | range(1, 4) | 1, 2, 3 |
| Step Negative | range(5, 0, -1) | 5, 4, 3, 2, 1 |

## 🧩 17. Using range() Without Loops

You can even store or use it directly in lists or tuples 👇

nums = list(range(0, 10, 3))

print(nums)

✅ **Output:**

[0, 3, 6, 9]

## 📦 18. Real-World Example

You can use range() for:

- Generating IDs
- Automating reports
- Running repetitive tests
- Creating pagination in apps

Example:

for page in range(1, 6):

  print("Downloading page", page)

✅ **Output:**

Downloading page 1

Downloading page 2

Downloading page 3

Downloading page 4

Downloading page 5

## ⚠️ 19. Common Mistakes

❌ Forgetting that stop value is **excluded**

```
for i in range(1, 5):
    print(i)
```

➡️ Prints only up to **4**, not 5.

❌ Using step = 0 (invalid)

```
for i in range(0, 10, 0):
    print(i)
```

❌ **Error:** ValueError: range() arg 3 must not be zero

## 🗒️ 20. Key Points to Remember

✔️ range() is mostly used in **for loops**
✔️ Stop value is **excluded**
✔️ Can work with **positive, negative, or step values**
✔️ Can convert to **list** using list(range(...))
✔️ Cannot have step = 0

## 🏁 21. Practice Questions

1. Print numbers from 1 to 10 using range().
2. Print all even numbers from 2 to 20.
3. Print numbers from 10 down to 1.
4. Generate a list of numbers from 5 to 50 with step 5.
5. Create a program to print multiplication table of any number entered by the user.
6. Create a list of all odd numbers between 1 and 30 using range().
7. Print squares of numbers from 1 to 10 using a for loop and range().

## 🎯 22. Summary

| Concept | Example | Output |
|---|---|---|
| Default start | range(5) | 0,1,2,3,4 |
| Start & Stop | range(2,6) | 2,3,4,5 |
| Step | range(0,10,2) | 0,2,4,6,8 |
| Reverse | range(10,0,-1) | 10→1 |
| Convert to List | list(range(1,6)) | [1,2,3,4,5] |

Day 20: Break and Continue Statements

# 🧠 Day 20: Break and Continue Statements in Python

## 🔍 1. Introduction

When you work with loops (like **for** and **while**), sometimes you may want to:

- **Stop** the loop completely under a certain condition 🛑
- Or **skip** one particular iteration and continue with the next one 🔁

Python gives us two powerful keywords for this:

- **break** → stops the loop completely
- **continue** → skips the current iteration and moves to the next

Let's learn both with clear examples and real-life logic 👇

## 🔸 2. The break Statement

The **break** statement is used to **terminate (stop)** a loop immediately — even if the loop condition is still true.

### 🧩 Syntax

```
for variable in sequence:
    if condition:
        break
    # code block
```

## 💡 3. Example 1 — Using break in a for loop

```
for i in range(1, 10):
    if i == 6:
        break
    print(i)
```

✅ **Output:**

```
1
2
3
4
5
```

🧠 **Explanation:**

- Loop runs from 1 to 9.
- When i becomes 6 → the condition i == 6 is True → loop stops immediately.
- The rest (6,7,8,9) are not printed.

## 💡 4. Example 2 — Using break in a while loop

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

✅ **Output:**

1

2

3

4

### Explanation:

- The loop stops when i equals 5.
- The break exits the loop right away.

## 🧠 5. Real-Life Analogy for break

Imagine you are searching for your lost key 🔑 in a drawer full of items.

You check each item one by one:

- If the item is not the key → keep checking.
- **If the item is the key → stop searching immediately!**
  → That's what **break** does — stops the process once the goal is found.

## 💻 6. Example 3 — Stop searching for a number

```
numbers = [2, 4, 6, 8, 10, 12]


for n in numbers:
    if n == 8:
        print("Found number:", n)
        break
    print("Checking number:", n)
```

✅ **Output:**

Checking number: 2

Checking number: 4

Checking number: 6

Found number: 8

## 💡 7. Example 4 — Nested Loops with break

```
for i in range(1, 4):
    for j in range(1, 4):
        if j == 2:
            break
        print(i, j)
```

✅ **Output:**

```
1 1
2 1
3 1
```

### Explanation:

- Inner loop breaks when j == 2.
- But the outer loop continues — break only affects the **innermost** loop.

## 🔷 8. The continue Statement

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

Unlike break, it doesn't stop the loop entirely — it just **skips that one turn**.

## 🧩 9. Syntax

```
for variable in sequence:
    if condition:
        continue
    # code block
```

## 💡 10. Example 1 — Using continue in a for loop

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

```
    print(i)
```

✅ **Output:**

1

2

4

5

### Explanation:

- When i == 3, the continue statement skips printing 3.
- The loop continues with 4, 5.

## 💡 11. Example 2 — Using continue in a while loop

```
i = 0

while i < 5:

    i += 1

    if i == 3:

        continue

    print(i)
```

✅ **Output:**

1

2

4

5

## 🧠 12. Real-Life Analogy for continue

Imagine you're taking attendance in class 🎓
If a student is **absent**, you **skip their name** and continue marking others.
That's exactly what continue does — skips that particular case and keeps going.

## 💡 13. Example 3 — Print only even numbers

```
for i in range(1, 10):
    if i % 2 != 0:
        continue
    print(i)
```

✅ **Output:**

```
2
4
6
8
```

### Explanation:

- When the number is odd (i % 2 != 0), skip it.
- Print only even numbers.

## 💡 14. Example 4 — Skip a particular item in a list

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


for fruit in fruits:
    if fruit == "mango":
        continue
    print("I like", fruit)
```

✅ **Output:**

```
I like apple
I like banana
I like orange
```

## 🔁 15. Example — Using break and continue together

```
for i in range(1, 10):
    if i == 5:
```

```
    continue  # skips 5
  if i == 8:
    break    # stops loop at 8
  print(i)
```

✅ **Output:**

1

2

3

4

6

7

**Explanation:**

- Skips printing 5 (continue).
- Stops the loop when i becomes 8 (break).

## ⚙️ 16. Loop Flow Comparison

| Keyword | What it Does | Example |
|---|---|---|
| break | Stops the loop completely | Exit search after finding item |
| continue | Skips the current iteration | Skip odd numbers in loop |

## 💡 17. Example — Searching in a List

```
numbers = [10, 20, 30, 40, 50]


for num in numbers:
  if num < 30:
```

```
    continue  # skip small numbers
  print("Number:", num)
  if num == 40:
    break  # stop when 40 found
```

✅ **Output:**

Number: 30

Number: 40

## 🧩 18. Example — Loop with User Input

```
while True:
  name = input("Enter your name (or 'stop' to quit): ")
  if name == "stop":
    break
  if name == "":
    continue
  print("Hello,", name)
```

✅ **Output:**

Enter your name (or 'stop' to quit): Naveen

Hello, Naveen

Enter your name (or 'stop' to quit):

Enter your name (or 'stop' to quit): stop

### Explanation:

- Empty input → skipped using continue.
- When user types 'stop' → loop ends with break.

## ⚠️ 19. Common Mistakes

❌ Forgetting to increment variable in a while loop → causes **infinite loop**
✅ Always increase/decrease counter before or after using continue.

❌ Using break outside a loop → causes an **error**
✅ break and continue only work **inside loops**.

## 🧾 20. Key Points to Remember

✔ break → exits the loop completely
✔ continue → skips current iteration
✔ Works in both for and while loops
✔ Use break when goal is achieved early
✔ Use continue when you need to skip certain data

## 🏁 21. Practice Questions

1. Print numbers from 1 to 10 but stop when the number reaches 7.
2. Print numbers 1–10 but skip 5 using continue.
3. Write a program to find the first multiple of 7 in a list and stop after finding it.
4. Ask users for names in a loop — skip empty names and stop when "exit" is entered.
5. Use both break and continue to print only even numbers below 20 and stop when you reach 16.

## 🎯 22. Summary

| Statement | Meaning | Behavior |
|---|---|---|
| break | Exit loop | Ends loop immediately |
| continue | Skip iteration | Goes to next iteration |
| Works in | for, while loops | Both |

✅ **In short:**

- **break** → completely stops the loop 🚫
- **continue** → skips a turn and continues 🔁

They make your loops **smarter and more flexible**, allowing fine control over how and when your code runs.

Day 21: Pass Statement

# 🧠 Day 21: Pass Statement in Python

## 🔍 1. Introduction

Sometimes in programming, we write code structures (like loops, functions, or conditionals) but don't yet know what logic to put inside.

In such cases, Python doesn't allow an *empty block* — it gives an **error** if you leave it blank.

To handle this, Python provides a special statement called **pass**.
It **does nothing** — it's a **placeholder** for future code.

Think of it as saying to Python:

"I'll fill in the details later — for now, just skip this part."

## 💡 2. Definition

**pass** is a null statement in Python.
It is used when a statement is syntactically required, but you don't want to execute any code.

## 🧩 3. Syntax

```
pass
```

It's that simple! 😄
No conditions, no loops — just a single word.

## 🎯 4. Why We Use pass

✔ To **avoid syntax errors** when leaving a block empty.
✔ To **create placeholders** for future code.
✔ To **maintain code structure** while designing complex programs.

## 💻 5. Example 1 — Empty If Statement

```
x = 10


if x > 5:
    pass  # placeholder for future logic
else:
    print("x is less than or equal to 5")
```

### 🧠 Explanation:

- The condition x > 5 is true.
- Since the if block only has pass, **Python skips it**.
- Nothing happens, but no error occurs.

### ✅ Output:

(nothing happens)

## 💻 6. Example 2 — Empty Loop

```
for i in range(5):
    pass  # loop runs but does nothing

print("Loop completed.")
```

### 🧠 Explanation:

- The loop iterates 5 times.
- Each time, Python encounters pass → skips execution.
- Then prints “Loop completed.”

### ✅ Output:

Loop completed.

## 💻 7. Example 3 — Empty Function

```
def greet():
    pass  # function defined but not implemented yet

greet()

print("Function executed successfully.")
```

### 🧠 Explanation:

- The function greet() exists, but has no body logic.
- Using pass allows it to compile and run without errors.

### ✅ Output:

Function executed successfully.

## 💻 8. Example 4 — Empty Class

```
class Person:
    pass  # will add attributes and methods later

print("Class created successfully.")
```

### 🧠 Explanation:

- You can define a class structure without adding any code.
- The pass statement prevents syntax errors.

✅ **Output:**

```
Class created successfully.
```

## 💻 9. Example 5 — Inside While Loop

```
count = 0

while count < 3:
    pass  # just a placeholder
    count += 1

print("While loop completed.")
```

✅ **Output:**

```
While loop completed.
```

## 🔁 10. Difference Between pass, continue, and break

| Statement | Purpose | Execution |
|---|---|---|
| **pass** | Does nothing | Skips and continues execution |
| **continue** | Skips current iteration | Moves to next iteration |
| **break** | Exits the loop completely | Terminates loop |

### 🧩 Example Comparison

```
for i in range(3):
    if i == 1:
        pass     # does nothing
    if i == 2:
        continue  # skips current iteration
    print(i)
```

✅ **Output:**

0

1

Explanation:

- When i == 1, pass does nothing → continues normally.
- When i == 2, continue skips the print.

## 🌍 11. Real-Life Analogy

Think of **pass** like an **empty note** in your notebook 📘:

✏️ You write the heading **"To-Do Tomorrow"**
But you don't write any tasks yet.
You just leave it blank for now — you'll fill it later.

Python's pass works the same way —
It keeps your code **syntactically correct** but **functionally empty**.

## ⚙️ 12. Common Use Cases

✅ In **unfinished functions or classes**
✅ In **loops or conditionals** where logic is not decided yet
✅ To **create placeholders** while designing complex code structure
✅ In **try-except blocks** to skip errors temporarily

## ⚠️ 13. Common Mistakes

❌ Leaving code blocks empty:

```
if True:

    # empty block

# This will cause an IndentationError!
```

✅ Correct way:

```
if True:

    pass
```

## 🧾 14. Key Points to Remember

✔️ pass is a **null operation** — does nothing
✔️ Used to **avoid syntax errors** in empty code blocks
✔️ Helps maintain **program structure**
✔️ Common in **development and prototyping**

## 💪 15. Practice Questions

1. Write a program with an empty function using pass.
2. Create a class named Student using pass (placeholder).
3. Use pass inside an if statement where no action is required.
4. Use pass inside a while loop that runs 3 times.
5. Demonstrate the difference between pass, continue, and break.

## 🏁 16. Summary

| Keyword | Meaning | Action |
|---|---|---|
| pass | Do nothing | Placeholder statement |
| continue | Skip current iteration | Moves to next loop cycle |
| break | Stop loop | Exits loop immediately |

✅ **In short:**
pass is Python's way of saying —

"I know this block should exist, but I'm not ready to fill it yet." 😄

Day 22: Lists - Introduction and Operations

# 🧠 Day 22: Lists — Introduction and Operations

## 🔍 1. Introduction

A **list** in Python is a **collection of multiple items** (like numbers, strings, or even other lists) stored in a **single variable**.

Think of a **list** as a **shopping list** 🛒:
You can keep all your items (milk, bread, eggs, etc.) together in one place.

Lists allow you to:

- Store multiple values together
- Access them easily using **indexes**
- Modify or remove them whenever needed

## 💡 2. Definition

A **List** is a **mutable**, **ordered** collection of elements enclosed within **square brackets [ ]**.

**Mutable** means: you can change, add, or remove items after creating the list.

## 🧩 3. Syntax

list_name = [item1, item2, item3, ...]

### Example:

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

✅ Here,

- fruits → list name
- "apple", "banana", "cherry" → list elements

## 🧮 4. Examples

### Example 1 — Creating a List

numbers = [10, 20, 30, 40]

print(numbers)

✅ Output:

[10, 20, 30, 40]

### Example 2 — Mixed Data Types

my_list = [25, "Python", 3.14, True]

print(my_list)

✅ Output:

[25, 'Python', 3.14, True]

### Example 3 — Nested List

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

print(nested)

✅ Output:

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

## 🧭 5. Accessing List Elements (Indexing)

Python lists are **indexed**, meaning every element has a position number.

| Element | Index |
|---|---|
| "apple" | 0 |
| "banana" | 1 |
| "cherry" | 2 |

Example:

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

print(fruits[0])  # first item

print(fruits[2])  # third item

✅ Output:

apple

cherry

## 🔄 6. Negative Indexing

Python allows **negative indexes** to access elements from the **end**.

| Element | Index | Negative Index |
|---|---|---|
| "apple" | 0 | -3 |
| "banana" | 1 | -2 |
| "cherry" | 2 | -1 |

Example:

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

print(fruits[-1])  # last element

✅ Output:

cherry

## ✂️ 7. Slicing a List

You can extract a **portion** of a list using slicing.

### Syntax:

list[start:end]

(End index is **not included**.)

### Example:

numbers = [10, 20, 30, 40, 50, 60]

print(numbers[1:4])

✅ Output:

[20, 30, 40]

## ⚙️ 8. List Operations

Python provides many built-in operations for lists.
Let's explore them one by one 👇

### 🧩 Concatenation (+)

Join two lists together.

a = [1, 2, 3]

b = [4, 5]

print(a + b)

✅ Output:

[1, 2, 3, 4, 5]

### 🧩 Repetition (*)

Repeat list elements.

fruits = ["apple", "banana"]

print(fruits * 2)

✅ Output:

['apple', 'banana', 'apple', 'banana']

### 🧩 Membership (in / not in)

Check if an item exists in the list.

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

print("apple" in fruits)

print("mango" not in fruits)

✅ Output:

True

True

## 🔁 9. Modifying List Elements

Lists are **mutable** → you can change their contents.

Example:

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

fruits[1] = "mango"

print(fruits)

✅ Output:

['apple', 'mango', 'cherry']

## ➕ 10. Adding Elements to a List

Using append() — adds item at the end

fruits = ["apple", "banana"]

fruits.append("cherry")

print(fruits)

✅ Output:

['apple', 'banana', 'cherry']

Using insert() — adds item at a specific position

```
fruits.insert(1, "mango")

print(fruits)
```

✅ Output:

```
['apple', 'mango', 'banana', 'cherry']
```

### Using extend() — adds elements from another list

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

b = [4, 5]

a.extend(b)

print(a)
```

✅ Output:

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

## — 11. Removing Elements from a List

### Using remove() — removes specific value

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

fruits.remove("banana")

print(fruits)
```

✅ Output:

```
['apple', 'cherry']
```

### Using pop() — removes by index

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

fruits.pop(1)

print(fruits)
```

✅ Output:

```
['apple', 'cherry']
```

### Using del — delete by index or entire list

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

del fruits[0]

print(fruits)
```

✅ Output:

```
['banana', 'cherry']
```

Using clear() — removes all elements

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

fruits.clear()

print(fruits)
```

✅ Output:

```
[]
```

## 📊 12. Useful List Functions

| Function | Description | Example | Output |
|---|---|---|---|
| len() | Returns number of items | len([10,20,30]) | 3 |
| max() | Returns largest element | max([4,7,2]) | 7 |
| min() | Returns smallest element | min([4,7,2]) | 2 |
| sum() | Adds all elements | sum([1,2,3]) | 6 |
| sorted() | Returns sorted list | sorted([3,1,2]) | [1,2,3] |

## 🧠 13. Looping Through a List

Example:

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

for f in fruits:

print(f)

✅ Output:

apple

banana

cherry

## 🌍 14. Real-Life Analogy

Imagine you have a **to-do list** 🗒️:

1. Wake up

2. Brush

3. Study

4. Exercise

You can:

- Add more tasks (append)
- Remove tasks (remove)
- Change order (insert)
- Check how many tasks (len)

Python lists behave just like that — flexible, organized, and easy to update!

## ⚠️ 15. Common Mistakes

❌ Accessing out-of-range index:

fruits = ["apple", "banana"]

print(fruits[3])  # Error: IndexError

✅ Always check list length using len(fruits) before accessing.

## 🗒️ 16. Key Points to Remember

✔️ Lists are **ordered** and **mutable**
✔️ Created using **square brackets [ ]**

✔ Can store **different data types**
✔ Support **indexing, slicing, and looping**
✔ Support **adding, removing, sorting** and more

## 💪 17. Practice Questions

1. Create a list of 5 cities and print them.
2. Change the 3rd city in the list to a new name.
3. Add a new city to the end using append().
4. Remove the first city from the list.
5. Print the total number of cities.
6. Combine two lists of numbers and print the result.
7. Print the even numbers from a list using a loop.

## 🏁 18. Summary

| Feature | Description |
|---|---|
| **Mutable** | You can modify list contents |
| **Ordered** | Items have fixed positions |
| **Index-based** | Access using indexes |
| **Heterogeneous** | Supports multiple data types |
| **Syntax** | [item1, item2, ...] |

Day 23: List Methods

# 🧠 Day 23: List Methods in Python

## 🔍 1. Introduction

You already know what a **list** is — a collection of multiple items stored in one variable.

Now imagine your **list** as a **toolbox** 🧰 — it comes with several **built-in tools (methods)** to:

- Add items 🧩
- Remove items ❌
- Sort items 🔢
- Count items 🔁
- Copy lists 📋

These tools make lists **powerful and flexible**.
Let's explore all major **list methods** step by step 👇

## 🧩 2. What is a Method?

A **method** is a **function associated with an object** (like a list).

It performs specific actions on that object.
You call it using **dot notation**:

list_name.method_name(arguments)

Example:

fruits.append("mango")

## ⚙️ 3. Commonly Used List Methods

Let's learn all the important ones with examples, explanations, and use cases 👇

## 🍎 append() — Add an item to the end of the list

Syntax:

list.append(item)

Example:

fruits = ["apple", "banana"]

fruits.append("cherry")

print(fruits)

✅ Output:

['apple', 'banana', 'cherry']

🟦 Explanation:

Adds the new element **at the end** of the list.
Think of it like adding a new item at the **bottom of your shopping list**.

## 🍌 insert() — Add an item at a specific position

Syntax:

list.insert(index, item)

Example:

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

fruits.insert(1, "mango")

print(fruits)

✅ Output:

['apple', 'mango', 'banana', 'cherry']

🟦 Explanation:

Inserts "mango" at index 1.
It shifts other elements to the right.

## 🍒 extend() — Add multiple items from another list

Syntax:

list.extend(iterable)

Example:

fruits = ["apple", "banana"]

tropical = ["mango", "pineapple"]

fruits.extend(tropical)

print(fruits)

✅ Output:

['apple', 'banana', 'mango', 'pineapple']

🟦 Explanation:

Joins two lists together — like merging two playlists 🎶.

## 🍉 remove() — Remove a specific item

Syntax:

list.remove(item)

Example:

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

fruits.remove("banana")

print(fruits)

✅ Output:

['apple', 'cherry']

⚠️ Note:

If the item doesn't exist, it will raise a **ValueError**.

## 🍇 pop() — Remove an item by index

Syntax:

list.pop(index)

Example:

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

fruits.pop(1)

print(fruits)

✅ Output:

['apple', 'cherry']

🟦 Explanation:

Removes the item **at index 1** (i.e., "banana").
If no index is provided → removes the **last item**.

## 🍓 clear() — Remove all items

Syntax:

list.clear()

Example:

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

fruits.clear()

print(fruits)
```

✅ Output:

```
[]
```

🟦 Explanation:

Empties the list but keeps it ready for reuse.

## 🍋 index() — Find the position of an item

Syntax:

```
list.index(item)
```

Example:

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

pos = fruits.index("banana")

print(pos)
```

✅ Output:

```
1
```

🟦 Explanation:

Returns the **first occurrence index** of the given item.

## 🫐 count() — Count how many times an item appears

Syntax:

```
list.count(item)
```

Example:

```
numbers = [1, 2, 2, 3, 2]

print(numbers.count(2))
```

✅ Output:

3

🟦 Explanation:

Counts how many times “2” appears in the list.

## 🍑 sort() — Sort the list in ascending order

Syntax:

list.sort()

Example:

numbers = [5, 1, 4, 2]

numbers.sort()

print(numbers)

✅ Output:

[1, 2, 4, 5]

🟦 Explanation:

Sorts the list in **ascending (A–Z / 0–9)** order.

To sort in **descending order**, use:

numbers.sort(reverse=True)

✅ Output:

[5, 4, 2, 1]

## 🍍 🔟 reverse() — Reverse the order of elements

Syntax:

list.reverse()

Example:

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

fruits.reverse()

print(fruits)

✅ Output:

['cherry', 'banana', 'apple']

🟦 Explanation:

Reverses the order of items — last becomes first.

## 🍎 1 copy() — Create a shallow copy of the list

Syntax:

new_list = list.copy()

Example:

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

new_fruits = fruits.copy()

print(new_fruits)

✅ Output:

['apple', 'banana', 'cherry']

🟦 Explanation:

Makes a **new copy** of the original list (not linked).

## 🍏 1 len() — Find total number of elements

Syntax:

len(list)

Example:

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

print(len(fruits))

✅ Output:

3

## 🍏 1 sum() — Sum of all numeric elements

Syntax:

sum(list)

Example:

numbers = [10, 20, 30]

print(sum(numbers))

✅ Output:

60

## 🍋 1 sorted() — Returns a new sorted list (without changing original)

Example:

numbers = [5, 2, 9, 1]

sorted_list = sorted(numbers)

print(sorted_list)

print(numbers)

✅ Output:

[1, 2, 5, 9]

[5, 2, 9, 1]

🟦 Explanation:

sorted() gives a **new list**, sort() modifies the **original**.

## 🧮 1 max(), min()

Example:

numbers = [10, 20, 5, 40]

print(max(numbers)) # largest

print(min(numbers)) # smallest

✅ Output:

40

5

## 🧠 1 Example Summary Table

| Method | Description | Example | Output |
|---|---|---|---|
| append() | Adds item to end | [1,2].append(3) | [1,2,3] |
| insert() | Inserts at index | [1,3].insert(1,2) | [1,2,3] |
| extend() | Joins lists | [1,2].extend([3,4]) | [1,2,3,4] |
| remove() | Removes value | [1,2,3].remove(2) | [1,3] |
| pop() | Removes by index | [1,2,3].pop(1) | [1,3] |
| clear() | Empties list | [1,2].clear() | [] |
| index() | Finds position | [1,2,3].index(2) | 1 |
| count() | Counts elements | [1,2,2].count(2) | 2 |
| sort() | Sorts ascending | [3,1,2].sort() | [1,2,3] |
| reverse() | Reverses order | [1,2,3].reverse() | [3,2,1] |
| copy() | Copies list | a.copy() | New list |
| len() | Length | len([1,2]) | 2 |
| sum() | Sum of numbers | sum([1,2,3]) | 6 |
| max() | Largest value | max([1,9,5]) | 9 |
| min() | Smallest value | min([1,9,5]) | 1 |

## 🌍 17. Real-Life Analogy

Imagine you're managing a **grocery list** 🛒:

- Add an item (append)
- Insert at top (insert)
- Remove an item (remove)
- Clear the list (clear)
- Count duplicates (count)
- Sort alphabetically (sort)
- Reverse the order (reverse)

These methods make your "grocery list" — or any **Python list** — fully manageable and flexible.

## ⚠️ 18. Common Mistakes

❌ Using = to copy lists (creates reference)

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

b = a  # both point to same list

b.append(4)

print(a)  # a is also changed
```

✅ Use copy() instead:

```
b = a.copy()

b.append(4)

print(a)  # a is safe
```

## 🧾 19. Key Points to Remember

✔️ List methods modify lists **in place** (except sorted()).
✔️ Lists are **mutable** — changes affect original data.
✔️ copy() creates a **shallow copy**, not a new independent object.
✔️ Always check **index validity** before pop() or remove().

## 💪 20. Practice Questions

1. Create a list of fruits and perform all basic methods (append, insert, remove, pop).
2. Write a program to count how many times "apple" appears in a list.
3. Sort a list of numbers in ascending and descending order.
4. Reverse a list and display both original and reversed lists.
5. Create two lists, combine them using extend() and append(). Compare results.
6. Copy a list using copy() and prove that they are independent.

## 🏁 21. Summary

| Feature | Description |
|---|---|
| **Mutable** | Can be changed after creation |
| **Methods** | Built-in tools to modify and manage lists |
| **Most Used** | append(), remove(), pop(), sort(), reverse(), count() |
| **Copy Caution** | Use copy() to avoid linked changes |

✅ **In short:**
**List methods** make your lists dynamic, flexible, and powerful — helping you add, remove, sort, and manage data efficiently, just like managing your daily to-do or shopping list! 🛍️

Day 24: Tuples - Introduction and Operations

# 🧠 Day 24: Tuples – Introduction and Operations in Python

## 🔍 1. Introduction

In Python, a **tuple** is a **collection data type** just like a list — it can store **multiple items** in a **single variable**.
But unlike lists, **tuples are immutable**, meaning **once created, you cannot change, add, or remove items**.

Think of a tuple like a **sealed box** 📦 — once you pack items and seal it, you can only **look inside** but not **change anything** inside.

## 💡 2. Definition

A **tuple** is an **ordered**, **immutable** collection that can contain **elements of different data types**.

## 🧩 3. Syntax

my_tuple = (item1, item2, item3)

✅ You use **round brackets ()** instead of square brackets [].

## 💻 4. Example – Creating a Tuple

fruits = ("apple", "banana", "cherry")

print(fruits)

✅ **Output:**

('apple', 'banana', 'cherry')

## 🧠 5. Tuple Characteristics

| Property | Description |
|---|---|
| **Ordered** | Items have a defined order |
| **Immutable** | Cannot be changed after creation |
| **Allows Duplicates** | Can contain duplicate values |
| **Can store mixed data types** | Example: (1, "apple", True, 2.5) |

## 💻 6. Example – Mixed Data Types

my_data = (101, "Naveen", 85.6, True)

print(my_data)

✅ **Output:**

(101, 'Naveen', 85.6, True)

## 🧠 7. Creating a Tuple Without Parentheses

You can create a tuple without parentheses by just separating values with commas.

numbers = 1, 2, 3, 4

print(numbers)

print(type(numbers))

✅ **Output:**

(1, 2, 3, 4)

<class 'tuple'>

## 💡 8. Creating a Single-Element Tuple

⚠️ Be careful — to create a **tuple with one element**, you must add a **comma** at the end.

single_tuple = ("apple",)

print(type(single_tuple))

✅ **Output:**

<class 'tuple'>

Without the comma:

not_a_tuple = ("apple")

print(type(not_a_tuple))

✅ **Output:**

<class 'str'>

## 🎯 9. Accessing Tuple Elements

You can access elements using **indexing** (like lists).

fruits = ("apple", "banana", "cherry")

print(fruits[0])  # first element

print(fruits[1])  # second element

print(fruits[-1])  # last element

✅ **Output:**

apple

banana

cherry

## 🧩 10. Slicing Tuples

You can extract a range of elements using slicing:

```
numbers = (10, 20, 30, 40, 50, 60)

print(numbers[1:4])   # elements from index 1 to 3

print(numbers[:3])    # first three elements

print(numbers[3:])    # from index 3 to end
```

✅ **Output:**

(20, 30, 40)

(10, 20, 30)

(40, 50, 60)

## 🔁 11. Looping Through Tuples

You can loop through tuple elements using a **for loop**.

```
colors = ("red", "green", "blue")


for color in colors:
    print(color)
```

✅ **Output:**

red

green

blue

## 🚫 12. Immutability – You Cannot Change Tuples

Tuples **cannot** be modified (no add, remove, or replace).

```
numbers = (1, 2, 3)

numbers[0] = 10   # ❌ This will cause an error
```

✅ **Output:**

```
TypeError: 'tuple' object does not support item assignment
```

## 🔄 13. Workaround – Converting Tuple to List

If you really need to modify a tuple,
you can **convert it to a list**, make changes, and then **convert back**.

```
numbers = (1, 2, 3)

temp = list(numbers)  # convert to list

temp[0] = 10          # modify

numbers = tuple(temp) # convert back to tuple


print(numbers)
```

✅ **Output:**

```
(10, 2, 3)
```

## 🧮 14. Tuple Operations

### ➤ Concatenation

```
t1 = (1, 2, 3)

t2 = (4, 5, 6)

result = t1 + t2

print(result)
```

✅ **Output:**

```
(1, 2, 3, 4, 5, 6)
```

### ➤ Repetition

```
numbers = (1, 2)

print(numbers * 3)
```

✅ **Output:**

(1, 2, 1, 2, 1, 2)

### ➤ Membership

fruits = ("apple", "banana", "cherry")

print("banana" in fruits)  # True

print("mango" not in fruits)  # True

✅ **Output:**

True

True

## 🧮 15. Tuple Functions

| Function | Description | Example | Output |
|---|---|---|---|
| len() | Returns number of elements | len((1,2,3)) | 3 |
| min() | Returns smallest element | min((5,2,9)) | 2 |
| max() | Returns largest element | max((5,2,9)) | 9 |
| sum() | Returns sum of elements | sum((5,2,3)) | 10 |
| tuple() | Converts an iterable into tuple | tuple([1,2,3]) | (1,2,3) |
| count() | Counts occurrences | (1,2,1).count(1) | 2 |
| index() | Finds index of element | (1,2,3).index(2) | 1 |

## 🧩 16. Nested Tuples

Tuples can contain other tuples (like 2D data).

```
nested = ((1, 2), (3, 4), (5, 6))

print(nested[1])     # (3, 4)

print(nested[1][0])  # 3
```

✅ **Output:**

```
(3, 4)

3
```

## 🧠 17. Tuple Unpacking

You can assign tuple elements to variables directly.

```
person = ("Naveen", 25, "India")

name, age, country = person


print(name)

print(age)

print(country)
```

✅ **Output:**

```
Naveen

25

India
```

## 🌍 18. Real-Life Analogy

Imagine a **Train Ticket** 🎟️
Once issued, you **cannot change** the passenger name, date, or seat —
You can only **view** it.

That's how a **tuple** works — fixed and unchangeable after creation.

## ⚙️ 19. When to Use Tuples vs Lists

| Feature | Tuple | List |
|---|---|---|
| **Mutable** | ❌ No | ✅ Yes |
| **Faster** | ✅ Yes | ❌ No |
| **Use case** | Fixed data | Changing data |
| **Syntax** | () | [] |

## 📝 20. Practice Questions

1. Create a tuple with 5 numbers and find their sum.
2. Access the 2nd and last element of a tuple.
3. Write a program to check if a given value exists in a tuple.
4. Create two tuples and concatenate them.
5. Convert a list [1, 2, 3, 4] into a tuple.
6. Unpack a tuple ("Python", "AI", "ML") into 3 variables.

## 🏁 21. Summary

| Concept | Description |
|---|---|
| **Tuple** | Ordered, immutable collection |
| **Syntax** | my_tuple = (1, 2, 3) |
| **Access** | Indexing or slicing |
| **Change** | Not allowed |
| **Loop** | Yes, using for loop |
| **Use case** | Fixed data like coordinates, months, etc. |

✅ **In short:**
A **tuple** is like a **locked container** — you can **see** the data but **can't modify** it 🔒
Perfect for storing **constant or unchanging data**!

# 🧠 Day 25: Tuple Methods in Python

We already learned what **Tuples** are — immutable, ordered collections used to store multiple items.
Now, we'll explore the **methods (functions)** that we can apply on tuples to perform operations or get useful information.

Even though tuples **can't be modified**, Python still gives us a few helpful methods to **analyze and work with** them.

## 🔍 1. Introduction

Unlike lists, tuples have **very few built-in methods**, because they are **immutable** (unchangeable).

There are only **two direct methods**:

1. count()
2. index()

However, there are also **many useful built-in functions** (not methods) that work **with tuples**, such as len(), min(), max(), sum(), and sorted().

We'll learn **all of them** with examples and outputs. 💪

## 🧩 2. Tuple Methods Overview

| Method | Description |
|---|---|
| count(value) | Returns the number of times a value appears in the tuple |
| index(value) | Returns the index of the first occurrence of the value |

## 💻 3. Example Tuple

Let’s take one tuple to work with:

numbers = (10, 20, 30, 10, 40, 10)

## 🧠 4. The count() Method

### ➤ Purpose:

Returns how many times a particular element appears in the tuple.

### ➤ Syntax:

tuple_name.count(value)

### ➤ Example:

numbers = (10, 20, 30, 10, 40, 10)

print(numbers.count(10))

### ✅ Output:

3

### 🧩 Explanation:

- The number 10 appears **3 times** in the tuple.
- So, count(10) returns 3.

## 💡 5. Real-Life Example of count()

Imagine you’re checking how many times a student scored 90 marks in 5 tests:

scores = (85, 90, 76, 90, 90)

print("Number of 90s:", scores.count(90))

### ✅ Output:

Number of 90s: 3

## 🧠 6. The index() Method

### ➤ Purpose:

Returns the **index (position)** of the **first occurrence** of the given value.

### ➤ Syntax:

tuple_name.index(value)

### ➤ Example:

numbers = (10, 20, 30, 10, 40)

print(numbers.index(10))

### ✅ Output:

0

### 🧩 Explanation:

- The first 10 is found at **index 0**, so it returns 0.

### ⚠️ Important Note:

If the value doesn't exist in the tuple, Python throws an **error**.

numbers = (10, 20, 30)

print(numbers.index(50))  # ❌

### ✅ Output:

ValueError: tuple.index(x): x not in tuple

## 💡 7. Real-Life Example of index()

Imagine a tuple of months, and you want to find the index of "March":

months = ("January", "February", "March", "April")

position = months.index("March")

print("March is at index:", position)

### ✅ Output:

March is at index: 2

## 🔢 8. Using Optional Parameters with index()

The index() method can take **two optional parameters** — start and end — to limit the search range within the tuple.

### ➤ Syntax:

tuple_name.index(value, start, end)

➤ **Example:**

```
numbers = (10, 20, 30, 10, 40, 10)

print(numbers.index(10, 1))      # start searching from index 1

print(numbers.index(10, 2, 5))   # search between index 2 and 4
```

✅ **Output:**

```
3

3
```

## ⚙️ 9. Other Built-In Functions That Work with Tuples

Even though these aren't "methods" (they're global functions), they're very useful when working with tuples.

🔹 **len()** → Find Length of Tuple

```
numbers = (10, 20, 30, 40)

print(len(numbers))
```

✅ **Output:**

```
4
```

🔹 **min()** → Find Smallest Element

```
numbers = (10, 20, 5, 30)

print(min(numbers))
```

✅ **Output:**

```
5
```

🔹 **max()** → Find Largest Element

```
numbers = (10, 20, 5, 30)

print(max(numbers))
```

✅ **Output:**

```
30
```

🔹 **sum()** → Add All Elements

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

print(sum(numbers))
```

✅ **Output:**

```
60
```

### 🔹 **sorted()** → Sort Tuple Elements

This returns a **list** (not a tuple).

```
numbers = (40, 10, 30, 20)

print(sorted(numbers))
```

✅ **Output:**

```
[10, 20, 30, 40]
```

If you want it back as a tuple:

```
sorted_tuple = tuple(sorted(numbers))

print(sorted_tuple)
```

✅ **Output:**

```
(10, 20, 30, 40)
```

## 💡 10. Counting Occurrences of Words

```
words = ("python", "java", "python", "c++", "python")

print("Python appears:", words.count("python"), "times.")
```

✅ **Output:**

```
Python appears: 3 times.
```

## 💡 11. Finding Index of a Name

```
students = ("Naveen", "Rahul", "Priya", "Naveen")

print("First occurrence of Naveen:", students.index("Naveen"))
```

✅ **Output:**

```
First occurrence of Naveen: 0
```

## 🧠 12. Why Are There So Few Tuple Methods?

Because **tuples are immutable** —
Python does not allow modifying, adding, or deleting elements.

So methods like:

- append()
- remove()
- insert()
- pop()

...that we saw in **lists**, **don't exist** for tuples.

Instead, tuple methods focus on **analyzing**, not **editing**.

## 🌍 13. Real-Life Analogy

Think of a tuple like a **locked safe** 🔒

- You can **count** how many things are inside (count()).
- You can **check where a particular item is** (index()).
- But you **can't add or remove** anything inside.

## 📚 14. Summary Table

| Method/Function | Description | Example | Output |
|---|---|---|---|
| count(value) | Counts occurrences | (1,2,1).count(1) | 2 |
| index(value) | Finds first index | (1,2,3).index(2) | 1 |
| len() | Returns number of elements | len((1,2,3)) | 3 |
| min() | Smallest element | min((3,1,2)) | 1 |
| max() | Largest element | max((3,1,2)) | 3 |
| sum() | Sum of elements | sum((2,3,4)) | 9 |
| sorted() | Returns sorted list | sorted((3,1,2)) | [1,2,3] |

## 💪 15. Practice Questions

1. Write a program to count how many times a number appears in a tuple.
2. Find the index of “blue” in the tuple ("red", "green", "blue", "yellow").
3. Create a tuple of marks and find the total using sum().
4. Sort the tuple (50, 10, 40, 20) in ascending order.
5. Create a tuple of 5 numbers and find the smallest and largest using min() and max().

## 🏁 16. Summary

✅ Tuples have **only two methods** → count() and index()
✅ Other useful functions: len(), min(), max(), sum(), sorted()
✅ Tuples are **immutable** → you can’t add, remove, or modify
✅ Useful for **fixed, read-only data**

**In short:**

Tuples don’t let you *change* data — but they help you *analyze* it efficiently! 💡

Day 26: Sets - Introduction and Operations

# 🧠 Day 26: Sets — Introduction and Operations

## 🔹 What is a Set in Python?

A **Set** in Python is an **unordered collection** of **unique elements**.
This means:

- It **doesn’t allow duplicates**
- It **does not maintain order**
- It is **mutable** (you can add or remove items)
- It is defined using **curly braces {}** or the **set() constructor**

Think of a **Set** like a **bag of unique items** — no duplicates, and the order doesn’t matter.

## 🔹 Why Use Sets?

Sets are useful when:

- You want to store **unique values**.
- You want to perform **mathematical operations** like union, intersection, and difference.

- You need to **check membership** quickly.

## 🔹 Creating a Set

### ✅ Example 1: Using Curly Braces

```
fruits = {"apple", "banana", "cherry"}

print(fruits)
```

🟩 **Output:**

```
{'cherry', 'banana', 'apple'}
```

👉 Notice: The order might change — sets are **unordered**.

### ✅ Example 2: Using set() Constructor

```
numbers = set([1, 2, 3, 4])

print(numbers)
```

🟩 **Output:**

```
{1, 2, 3, 4}
```

### ✅ Example 3: Removing Duplicates Automatically

```
data = {1, 2, 2, 3, 4, 4, 5}

print(data)
```

🟩 **Output:**

```
{1, 2, 3, 4, 5}
```

👉 Sets **automatically remove duplicate values**.

## 🔹 Accessing Elements in a Set

You **cannot access elements by index** because sets are unordered.

But you can use a **loop** to access each item:

```
colors = {"red", "green", "blue"}

for color in colors:
    print(color)
```

🟩 **Output:**

green

red

blue

*(The order may vary)*

## 🔹 Checking Membership

You can use the in and not in operators:

```
animals = {"cat", "dog", "rabbit"}

print("dog" in animals)      # True

print("lion" not in animals)  # True
```

🟩 **Output:**

True

True

## 🔹 Adding Elements to a Set

### ✅ add() method

Used to add a single element.

```
numbers = {1, 2, 3}

numbers.add(4)

print(numbers)
```

🟩 **Output:**

{1, 2, 3, 4}

### ✅ update() method

Used to add **multiple elements** at once.

```
numbers = {1, 2, 3}

numbers.update([4, 5, 6])
```

print(numbers)

🟩 **Output:**

{1, 2, 3, 4, 5, 6}

## 🔹 Removing Elements from a Set

| Method | Description | Example | Error if not present? |
|---|---|---|---|
| remove() | Removes the specified item | set.remove(item) | ❌ Yes |
| discard() | Removes item if exists | set.discard(item) | ✅ No |
| pop() | Removes a random item | set.pop() | ✅ No |
| clear() | Removes all elements | set.clear() | ✅ No |

### ✅ Examples:

numbers = {1, 2, 3, 4, 5}

numbers.remove(3)

print(numbers)

numbers.discard(10)  # No error even if not found

numbers.pop()

print(numbers)

numbers.clear()

```
print(numbers)
```

🟩 **Output:**

```
{1, 2, 4, 5}

{2, 4, 5}

set()
```

## 🔹 Set Operations (Mathematical Operations)

Python sets support operations like in math.

### ✅ 1. Union (| or union())

Combines all elements from both sets (no duplicates).

```
a = {1, 2, 3}

b = {3, 4, 5}

print(a | b)

print(a.union(b))
```

🟩 **Output:**

```
{1, 2, 3, 4, 5}
```

### ✅ 2. Intersection (& or intersection())

Returns elements common to both sets.

```
print(a & b)

print(a.intersection(b))
```

🟩 **Output:**

```
{3}
```

### ✅ 3. Difference (- or difference())

Returns elements in one set but not in the other.

```
print(a - b)

print(b - a)
```

🟩 **Output:**

{1, 2}

{4, 5}

### ✅ 4. Symmetric Difference (^ or symmetric_difference())

Returns elements present in either set but not both.

print(a ^ b)

print(a.symmetric_difference(b))

🟩 **Output:**

{1, 2, 4, 5}

## 🔹 Set Comparison Operators

| Operator | Meaning |
|---|---|
| == | Checks if two sets are equal |
| != | Checks if two sets are not equal |
| > | Checks if first set is a superset |
| < | Checks if first set is a subset |
| >= | Checks if first set is equal or superset |
| <= | Checks if first set is equal or subset |

### ✅ Example:

a = {1, 2, 3}

b = {1, 2, 3, 4, 5}

print(a < b)  # True (subset)

print(b > a)  # True (superset)

## 🔹 Real-Life Example: Email Management System

Imagine you are creating a mailing system and want to remove duplicate email addresses.

emails = ["john@gmail.com", "alex@gmail.com", "john@gmail.com", "mark@gmail.com"]

unique_emails = set(emails)

print(unique_emails)

🟩 **Output:**

{'mark@gmail.com', 'alex@gmail.com', 'john@gmail.com'}

👉 Sets automatically removed the duplicates.

## 🔹 Key Points Summary

| Feature | Description |
|---|---|
| **Duplicates** | Not allowed |
| **Ordered** | No |
| **Mutable** | Yes |
| **Indexing/Slicing** | Not supported |
| **Use Case** | Storing unique data, mathematical set operations |
| **Syntax** | {element1, element2, ...} or set() |

## 🔹 Practice Exercise

#  Create two sets of students

python_students = {"Ravi", "Anu", "Raj", "Meena"}

excel_students = {"Raj", "Meena", "Karan", "Tina"}

```
#       Students learning both
print("Both:", python_students & excel_students)

#       Students learning only Python
print("Only Python:", python_students - excel_students)

#       All students
print("All:", python_students | excel_students)
```

🟩 **Expected Output:**

```
Both: {'Raj', 'Meena'}
Only Python: {'Anu', 'Ravi'}
All: {'Tina', 'Raj', 'Anu', 'Meena', 'Ravi', 'Karan'}
```

Day 27: Set Methods

# 🧠 Day 27: Set Methods in Python

## 🔹 Introduction

In the previous day (Day 26), we learned about **what sets are** and how to perform **basic operations** like union, intersection, and difference.

Today, we'll go one step further and learn about the **built-in methods** that Python provides for sets — these make working with sets **easier, faster, and more powerful** 💪

## 🧩 What Are Set Methods?

**Set methods** are pre-defined functions in Python that allow you to:

- Add or remove elements
- Combine sets

- Compare sets
- Perform mathematical operations easily

Let’s understand them one by one with **syntax, examples, and outputs** 👇

## 🔹 add()

🟦 **Purpose:** Adds a single element to a set.

📜 **Syntax:**

set.add(element)

✅ **Example:**

numbers = {1, 2, 3}

numbers.add(4)

print(numbers)

🟩 **Output:**

{1, 2, 3, 4}

💡 **Note:**
If the element already exists, nothing happens (no duplicates allowed).

## 🔹 update()

🟦 **Purpose:** Adds multiple elements (from list, tuple, or another set).

📜 **Syntax:**

set.update(iterable)

✅ **Example:**

fruits = {"apple", "banana"}

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

print(fruits)

🟩 **Output:**

{'banana', 'apple', 'cherry', 'mango'}

💡 **Note:**
You can also pass another set to update().

## 🔹 remove()

🟦 **Purpose:** Removes a specific element from the set.
❗ Raises an **error** if the element doesn't exist.

📜 **Syntax:**

```
set.remove(element)
```

✅ **Example:**

```
colors = {"red", "green", "blue"}

colors.remove("green")

print(colors)
```

🟩 **Output:**

```
{'red', 'blue'}
```

❌ **If element not found:**

```
colors.remove("yellow")
```

🟥 **Error:**

```
KeyError: 'yellow'
```

## 🔹 discard()

🟦 **Purpose:** Removes the specified element if it exists.
✅ No error if the element is not found.

📜 **Syntax:**

```
set.discard(element)
```

✅ **Example:**

```
animals = {"cat", "dog", "rabbit"}

animals.discard("dog")

animals.discard("lion")  # No error

print(animals)
```

🟩 **Output:**

{'cat', 'rabbit'}

## 🔹 pop()

🟦 **Purpose:** Removes and returns a **random element** from the set.

📜 **Syntax:**

set.pop()

✅ **Example:**

numbers = {1, 2, 3, 4, 5}

removed = numbers.pop()

print("Removed:", removed)

print("Remaining Set:", numbers)

🟩 **Output:**

Removed: 1

Remaining Set: {2, 3, 4, 5}

💡 **Note:**
Since sets are unordered, you **can't predict** which element will be removed.

## 🔹 clear()

🟦 **Purpose:** Removes **all elements** from the set.

📜 **Syntax:**

set.clear()

✅ **Example:**

data = {10, 20, 30}

data.clear()

print(data)

🟩 **Output:**

set()

## 🔹 copy()

🟦 **Purpose:** Returns a **shallow copy** (a duplicate) of the set.

📜 **Syntax:**

new_set = set.copy()

✅ **Example:**

a = {"x", "y", "z"}

b = a.copy()

b.add("w")

print("Original:", a)

print("Copied:", b)

🟩 **Output:**

Original: {'x', 'y', 'z'}

Copied: {'x', 'y', 'z', 'w'}

💡 **Note:**
a and b are now **different objects**.

## 🔹 union()

🟦 **Purpose:** Returns all unique elements from both sets.

📜 **Syntax:**

set1.union(set2)

✅ **Example:**

a = {1, 2, 3}

b = {3, 4, 5}

print(a.union(b))

🟩 **Output:**

{1, 2, 3, 4, 5}

## 🔹 intersection()

🟦 **Purpose:** Returns elements **common to both sets**.

📜 **Syntax:**

set1.intersection(set2)

✅ **Example:**

x = {1, 2, 3, 4}

y = {3, 4, 5, 6}

print(x.intersection(y))

🟩 **Output:**

{3, 4}

## 🔹 🔟 difference()

🟦 **Purpose:** Returns elements present in the **first set but not in the second**.

📜 **Syntax:**

set1.difference(set2)

✅ **Example:**

x = {1, 2, 3, 4}

y = {3, 4, 5, 6}

print(x.difference(y))

🟩 **Output:**

{1, 2}

## 🔹 1 symmetric_difference()

🟦 **Purpose:** Returns elements that are in **either set, but not both**.

📜 **Syntax:**

set1.symmetric_difference(set2)

✅ **Example:**

a = {1, 2, 3}

b = {3, 4, 5}

print(a.symmetric_difference(b))

🟩 **Output:**

{1, 2, 4, 5}

## 🔹 1 isdisjoint()

🟦 **Purpose:** Returns True if two sets have **no elements in common**.

📜 **Syntax:**

set1.isdisjoint(set2)

✅ **Example:**

a = {1, 2, 3}

b = {4, 5, 6}

print(a.isdisjoint(b))

🟩 **Output:**

True

## 🔹 1 issubset()

🟦 **Purpose:** Checks if all elements of one set exist in another.

📜 **Syntax:**

set1.issubset(set2)

✅ **Example:**

a = {1, 2}

b = {1, 2, 3, 4}

print(a.issubset(b))

🟩 **Output:**

True

## 🔹 1 issuperset()

🟦 **Purpose:** Checks if a set contains all elements of another set.

📜 **Syntax:**

set1.issuperset(set2)

✅ **Example:**

x = {1, 2, 3, 4}

y = {2, 3}

print(x.issuperset(y))

🟩 **Output:**

True

## 🔹 1 difference_update()

🟦 **Purpose:** Removes all elements of another set from the current set.

📜 **Syntax:**

set1.difference_update(set2)

✅ **Example:**

a = {1, 2, 3, 4}

b = {3, 4}

a.difference_update(b)

print(a)

🟩 **Output:**

{1, 2}

## 🔹 1 intersection_update()

🟦 **Purpose:** Updates the set to keep only items **present in both sets**.

📜 **Syntax:**

set1.intersection_update(set2)

✅ **Example:**

a = {1, 2, 3, 4}

b = {3, 4, 5, 6}

a.intersection_update(b)

print(a)

🟩 **Output:**

{3, 4}

## 🔹 1 symmetric_difference_update()

🟦 **Purpose:** Updates the set with items that are **not common** in both sets.

📜 **Syntax:**

set1.symmetric_difference_update(set2)

✅ **Example:**

a = {1, 2, 3}

b = {3, 4, 5}

a.symmetric_difference_update(b)

print(a)

🟩 **Output:**

{1, 2, 4, 5}

## 🧠 Quick Summary Table

| Method | Description |
|---|---|
| add() | Adds one element |
| update() | Adds multiple elements |
| remove() | Removes element (error if not found) |
| discard() | Removes element (no error) |
| pop() | Removes random element |
| clear() | Removes all elements |
| copy() | Returns shallow copy |
| union() | Combines both sets |
| intersection() | Common elements |
| difference() | Elements in one set only |
| symmetric_difference() | Uncommon elements |
| issubset() | Checks if one set is subset |
| issuperset() | Checks if one set is superset |
| isdisjoint() | Checks if no elements in common |
| difference_update() | Removes elements of another set |
| intersection_update() | Keeps only common elements |
| symmetric_difference_update() | Keeps only different elements |

## 💼 Real-World Example: Online Course Students

python_students = {"Ravi", "Anu", "Meena", "Raj"}

```
excel_students = {"Meena", "Raj", "Karan", "Tina"}


# Students enrolled in both
print("Both:", python_students.intersection(excel_students))


# Students enrolled in Python only
print("Only Python:", python_students.difference(excel_students))


# Students enrolled in either
print("All Students:", python_students.union(excel_students))
```

🟩 **Output:**

```
Both: {'Raj', 'Meena'}
Only Python: {'Anu', 'Ravi'}
All Students: {'Karan', 'Raj', 'Meena', 'Tina', 'Anu', 'Ravi'}
```

Day 28: Dictionaries - Introduction and Operations

# 🧠 Day 28: Dictionaries — Introduction and Operations in Python

## 🔹 1. Introduction to Dictionaries

Imagine you want to store a student's details like:

Name → "Naveen"

Age → 22

Course → "Python"

If you use a list or tuple, it becomes confusing to know what each value represents.
But Python provides a special data type called a **Dictionary** 🗂️ —

which stores **data in key–value pairs**.

👉 A **key** acts like a label (e.g., “name”)
👉 A **value** is the actual data (e.g., “Naveen”)

## 🧩 2. Definition

A **dictionary** is an **unordered**, **mutable**, and **indexed** collection of data in **key–value pairs**.

Each **key** in a dictionary must be **unique** and **immutable** (like strings, numbers, or tuples).
Each **value** can be of **any data type** (string, list, integer, etc.).

## 💻 3. Syntax

```
dictionary_name = {
   "key1": "value1",
   "key2": "value2",
   "key3": "value3"
}
```

## 🧠 4. Example

```
student = {
   "name": "Naveen",
   "age": 22,
   "course": "Python"
}


print(student)
```

✅ **Output:**

```
{'name': 'Naveen', 'age': 22, 'course': 'Python'}
```

## 🔍 5. Features of Dictionaries

| Feature | Description |
| --- | --- |
| **Unordered** | Elements are not stored in a specific order |
| **Mutable** | You can change, add, or remove elements |
| **Key–Value pairs** | Each entry has a unique key and corresponding value |
| **Indexed by keys** | You access elements using keys, not positions |

## 🧮 6. Accessing Dictionary Elements

You can access elements using the **key name**.

student = {"name": "Naveen", "age": 22, "course": "Python"}

print(student["name"])

print(student["age"])

✅ **Output:**

Naveen

22

## 🧠 7. Using get() Method

To avoid errors if the key doesn't exist, use the .get() method.

print(student.get("course"))

print(student.get("grade", "Not Found"))

✅ **Output:**

Python

Not Found

💡 **Tip:**
get() allows you to set a **default value** if the key doesn't exist.

## 🧱 8. Adding New Items

You can add a new key-value pair easily.

student["grade"] = "A"

print(student)

✅ **Output:**

{'name': 'Naveen', 'age': 22, 'course': 'Python', 'grade': 'A'}

## 🔄 9. Updating Existing Items

Use the key directly or the .update() method.

student["age"] = 23

student.update({"course": "Advanced Python"})

print(student)

✅ **Output:**

{'name': 'Naveen', 'age': 23, 'course': 'Advanced Python', 'grade': 'A'}

## ❌ 10. Removing Items

### 🔸 Using pop():

Removes a specific key and returns its value.

student.pop("grade")

print(student)

✅ **Output:**

{'name': 'Naveen', 'age': 23, 'course': 'Advanced Python'}

### 🔸 Using popitem():

Removes the **last inserted item** (Python 3.7+).

student.popitem()

print(student)

✅ **Output:**

{'name': 'Naveen', 'age': 23}

### 🔸 Using del:

Removes a specific item or the entire dictionary.

del student["age"]

print(student)

✅ **Output:**

{'name': 'Naveen'}

💣 Delete entire dictionary:

del student

### 🔸 Using clear():

Removes all items but keeps the dictionary structure.

student = {"name": "Naveen", "age": 22}

student.clear()

print(student)

✅ **Output:**

{}

## 🧩 11. Checking if a Key Exists

You can check using the in keyword.

student = {"name": "Naveen", "age": 22, "course": "Python"}

if "course" in student:

    print("Yes, 'course' key is present.")

✅ **Output:**

Yes, 'course' key is present.

## 🧠 12. Looping Through a Dictionary

### 🔹 Loop through keys:

```
for key in student:
    print(key)
```

✅ Output:

```
name
age
course
```

### 🔹 Loop through values:

```
for value in student.values():
    print(value)
```

✅ Output:

```
Naveen
22
Python
```

### 🔹 Loop through key-value pairs:

```
for key, value in student.items():
    print(key, ":", value)
```

✅ Output:

```
name : Naveen
age : 22
course : Python
```

## ⚙️ 13. Nested Dictionaries

You can create a dictionary **inside another dictionary**.

```
students = {
    "student1": {"name": "Naveen", "age": 22},
```

```
    "student2": {"name": "Raj", "age": 23}
}


print(students["student1"]["name"])
```

✅ **Output:**

```
Naveen
```

## 🧮 14. Dictionary Length

Use len() to count the number of key-value pairs.

```
student = {"name": "Naveen", "age": 22, "course": "Python"}

print(len(student))
```

✅ **Output:**

```
3
```

## 🔧 15. Copying a Dictionary

### 🔹 Using copy():

```
student_copy = student.copy()

print(student_copy)
```

✅ **Output:**

```
{'name': 'Naveen', 'age': 22, 'course': 'Python'}
```

### 🔹 Using dict():

```
student_clone = dict(student)

print(student_clone)
```

✅ **Output:**

```
{'name': 'Naveen', 'age': 22, 'course': 'Python'}
```

## 💼 16. Real-World Example

Let’s store product details in a store.

```
product = {
    "id": 101,
    "name": "Laptop",
    "brand": "Dell",
    "price": 65000,
    "in_stock": True
}

# Access product info
print("Product Name:", product["name"])
print("Price:", product["price"])

# Update price
product["price"] = 62000
print("Updated Price:", product["price"])

# Add discount info
product["discount"] = "10%"
print(product)
```

✅ **Output:**

```
Product Name: Laptop
Price: 65000
Updated Price: 62000
{'id': 101, 'name': 'Laptop', 'brand': 'Dell', 'price': 62000, 'in_stock': True, 'discount': '10%'}
```

## 🧠 17. Dictionary Methods Summary

| Method | Description |
|---|---|
| get() | Returns value for a key |
| keys() | Returns all keys |
| values() | Returns all values |
| items() | Returns key-value pairs |
| pop() | Removes key and returns its value |
| popitem() | Removes last item |
| update() | Adds or updates items |
| clear() | Removes all items |
| copy() | Copies dictionary |

## 🧾 18. Real-Life Analogy

Think of a **dictionary** like a real dictionary 📖:

- The **word** = key
- The **meaning** = value

When you search for a word (key), you find its meaning (value).
Similarly, in Python, when you search by key, you get its value instantly.

## ✅ 19. Key Takeaways

✔ A dictionary stores data in **key-value pairs**
✔ Keys are **unique and immutable**
✔ Values can be **any type**
✔ Supports **nesting, updating, and iteration**
✔ Extremely useful for **structured data representation**

## 💪 20. Practice Tasks

Create a dictionary of your favorite movie (title, year, director, rating).
Add a new key "genre" to it.

Update the "rating" value.
Remove one key using pop().
Loop through all keys and values and print them.

### ✅ In Short:

A Python Dictionary is the most powerful data type to store and manage structured, real-world data — fast, flexible, and human-readable.

## Day 29: Dictionary Methods

# 🧭 Day 29: Dictionary Methods in Python

### 🎯 Goal:

To learn how to use built-in **dictionary methods** in Python to efficiently manipulate, update, and retrieve data from dictionaries.

## 🧩 What are Dictionary Methods?

Dictionary methods are **predefined functions** that help you work with dictionaries — such as adding items, removing items, copying, clearing, and more.

Each method allows you to perform specific operations on dictionaries easily without manually looping or handling exceptions.

## 🧠 Commonly Used Dictionary Methods

Let’s go one by one 👇

### 🔹 1. dict.keys()

Returns all the **keys** of the dictionary.

🧩 Example:

```
student = {'name': 'Naveen', 'age': 21, 'course': 'Python'}

print(student.keys())
```

🎯 Output:

dict_keys(['name', 'age', 'course'])

✅ **Note:** It returns a view object — which updates automatically when the dictionary changes.

### 🔹 2. dict.values()

Returns all the **values** in the dictionary.

🧩 Example:

print(student.values())

🎯 Output:

dict_values(['Naveen', 21, 'Python'])

### 🔹 3. dict.items()

Returns each **key-value pair** as a tuple inside a list.

🧩 Example:

print(student.items())

🎯 Output:

dict_items([('name', 'Naveen'), ('age', 21), ('course', 'Python')])

✅ Useful when iterating through both keys and values.

### 🔹 4. dict.get(key, default)

Returns the **value** for a specific key.
If the key doesn't exist, it returns the **default value**.

🧩 Example:

print(student.get('age'))

print(student.get('city', 'Not Found'))

🎯 Output:

21

Not Found

✅ Safer than using student['city'], which would raise a KeyError.

### 🔹 5. dict.update(other_dict)

Updates the dictionary with key-value pairs from another dictionary.

🧩 Example:

```
student.update({'city': 'Hyderabad'})

print(student)
```

🎯 Output:

```
{'name': 'Naveen', 'age': 21, 'course': 'Python', 'city': 'Hyderabad'}
```

✅ If the key exists, it **overwrites** the old value.

### 🔹 6. dict.pop(key, default)

Removes the specified key and returns its value.
If the key is not found, returns the default value.

🧩 Example:

```
age = student.pop('age')

print(age)

print(student)
```

🎯 Output:

```
21

{'name': 'Naveen', 'course': 'Python', 'city': 'Hyderabad'}
```

### 🔹 7. dict.popitem()

Removes and returns the **last inserted key-value pair**.

🧩 Example:

```
last_item = student.popitem()

print(last_item)

print(student)
```

🎯 Output:

```
('city', 'Hyderabad')

{'name': 'Naveen', 'course': 'Python'}
```

✅ From Python 3.7+, dictionaries maintain insertion order.

## 🔹 8. dict.copy()

Creates a **shallow copy** of the dictionary.

🧩 Example:

```
copy_student = student.copy()

print(copy_student)
```

🎯 Output:

```
{'name': 'Naveen', 'course': 'Python'}
```

✅ Useful for avoiding changes to the original dictionary.

## 🔹 9. dict.clear()

Removes all items from the dictionary, leaving it empty.

🧩 Example:

```
student.clear()

print(student)
```

🎯 Output:

```
{}
```

## 🔹 10. dict.fromkeys(sequence, value)

Creates a new dictionary from a list of keys and assigns them all the same value.

🧩 Example:

```
subjects = ['Python', 'SQL', 'Power BI']

marks = dict.fromkeys(subjects, 0)

print(marks)
```

🎯 Output:

```
{'Python': 0, 'SQL': 0, 'Power BI': 0}
```

✅ Great for initializing multiple keys with the same default value.

## 🧮 Summary Table of Dictionary Methods

| Method | Description | Example |
|---|---|---|
| keys() | Returns all keys | dict.keys() |
| values() | Returns all values | dict.values() |
| items() | Returns all key-value pairs | dict.items() |
| get() | Returns value for key | dict.get('key') |
| update() | Adds or updates key-value pairs | dict.update({'a':1}) |
| pop() | Removes key and returns value | dict.pop('key') |
| popitem() | Removes last item | dict.popitem() |
| copy() | Returns a copy of the dict | dict.copy() |
| clear() | Empties the dictionary | dict.clear() |
| fromkeys() | Creates new dict with given keys | dict.fromkeys(['a','b'],0) |

## 💡 Real-Life Example:

🧩 Example — Student Marks Tracker

marks = {'Math': 85, 'Science': 90, 'English': 78}

# Add new subject

marks.update({'Python': 95})

```
# Get all subjects
print("Subjects:", marks.keys())

# Remove one subject
marks.pop('English')

# Total subjects
print("Subjects Count:", len(marks))

# Copy marks
copy_marks = marks.copy()
print("Copy of Marks:", copy_marks)
```

### 🎯 Output:

```
Subjects: dict_keys(['Math', 'Science', 'English', 'Python'])
Subjects Count: 3
Copy of Marks: {'Math': 85, 'Science': 90, 'Python': 95}
```

## 🧠 Key Points to Remember

✅ get() is safer than using [] for key access.
✅ popitem() removes the last inserted item.
✅ update() merges or overwrites key-value pairs.
✅ copy() helps avoid unwanted changes to the original.
✅ clear() empties the entire dictionary.

## 🏁 Conclusion:

By mastering **dictionary methods**, you can easily:

- Retrieve and manage data efficiently
- Avoid errors when keys are missing
- Write cleaner and shorter code

# 🧭 Day 30: Strings – Introduction and Operations

## 🎯 Goal:

To understand **what strings are**, how to **create**, **access**, and **manipulate** them using **various operations and methods** in Python.

## 🧠 What is a String in Python?

A **String** is a **sequence of characters** enclosed within **single quotes (‘ ’)**, **double quotes (“ ”)**, or **triple quotes (“”” ”””)**.

Strings are one of the **most commonly used data types** in Python — used for storing textual data such as names, messages, or sentences.

### 🔹 Examples of Strings

name = 'Naveen'

greeting = "Hello, World!"

paragraph = """This is

a multi-line

string."""

✅ Python allows **single (')**, **double (")**, **and triple (""" """)** quotes to define strings.
✅ Triple quotes can span **multiple lines**.

## ⚙️ How Strings Work Internally

- Strings are **immutable**, meaning once created, they **cannot be changed**.
- Any modification creates a **new string object** in memory.

## 🧩 Creating Strings

# Single-quoted

text1 = 'Python'

# Double-quoted

```
text2 = "Power BI"


# Triple-quoted (Multi-line)
text3 = """Learning Python
is fun!"""

print(text1)
print(text2)
print(text3)
```

🎯 **Output:**

```
Python
Power BI
Learning Python
is fun!
```

## 🧮 Accessing Characters in a String

We can access specific characters using **indexing**.

### 🔹 Example:

```
word = "Python"
print(word[0])   # First character
print(word[5])   # Last character
```

🎯 **Output:**

```
P
n
```

✅ Indexing starts from **0** (like lists and tuples).
✅ Negative indexing is also allowed.

### 🔹 Negative Indexing

```
word = "Python"

print(word[-1])  # Last character

print(word[-3])  # Third from last
```

🎯 **Output:**

```
n

h
```

## ✂️ Slicing Strings

We can extract a **portion of a string** using slicing syntax:

```
string[start:end:step]
```

### 🔹 Example:

```
text = "Python Programming"

print(text[0:6])    # From index 0 to 5

print(text[7:])     # From index 7 to end

print(text[:6])     # From start to index 5

print(text[::2])    # Every 2nd character

print(text[::-1])   # Reverse string
```

🎯 **Output:**

```
Python

Programming

Python

Pto rgamn

gnimmargorP nohtyP
```

✅ Slicing creates **new strings** (does not modify the original one).

## ⚡ String Concatenation and Repetition

### 🔹 Concatenation (+)

Used to **combine** two or more strings.

first = "Hello"

second = "Naveen"

message = first + " " + second

print(message)

🎯 **Output:**

Hello Naveen

### 🔹 Repetition (*)

Used to **repeat** a string multiple times.

word = "Hi "

print(word * 3)

🎯 **Output:**

Hi Hi Hi

## 🔠 String Length

Use the built-in len() function to get the number of characters.

text = "Python"

print(len(text))

🎯 **Output:**

6

## 🧮 Membership Operators in Strings

You can check if a substring **exists** in another string using in and not in.

text = "Learning Python is easy"

print("Python" in text)

print("Java" not in text)

🎯 **Output:**

True

True

## 🔤 Iterating Through a String

You can loop through each character in a string.

```
word = "Data"

for ch in word:
    print(ch)
```

🎯 **Output:**

D

a

t

a

## 💎 String Comparison

Strings can be compared using **comparison operators** (==, !=, <, >, etc.).

```
a = "apple"

b = "banana"


print(a == b)

print(a < b)    # Alphabetical order comparison
```

🎯 **Output:**

False

True

## 🧰 String Methods (Most Useful Ones)

Let’s go through the most frequently used **string methods**.

### 🔹 1. upper() – Convert to Uppercase

```
name = "naveen"

print(name.upper())
```

🎯 Output:

```
NAVEEN
```

### 🔹 2. lower() – Convert to Lowercase

```
name = "PYTHON"

print(name.lower())
```

🎯 Output:

```
python
```

### 🔹 3. title() – Capitalize Each Word

```
text = "learn python easily"

print(text.title())
```

🎯 Output:

```
Learn Python Easily
```

### 🔹 4. capitalize() – Capitalize First Letter

```
text = "hello world"

print(text.capitalize())
```

🎯 Output:

```
Hello world
```

### 🔹 5. strip() – Remove Whitespace

```
text = "   Naveen   "

print(text.strip())
```

🎯 Output:

```
Naveen
```

### 🔹 6. replace(old, new) – Replace Substring

```
text = "I love Java"
```

```
print(text.replace("Java", "Python"))
```

🎯 Output:

```
I love Python
```

### 🔹 7. split() – Split String into List

```
text = "Python,SQL,PowerBI"

print(text.split(","))
```

🎯 Output:

```
['Python', 'SQL', 'PowerBI']
```

### 🔹 8. join() – Join List into String

```
words = ['Python', 'is', 'fun']

print(" ".join(words))
```

🎯 Output:

```
Python is fun
```

### 🔹 9. find() – Find Index of Substring

```
text = "Learning Python"

print(text.find("Python"))
```

🎯 Output:

```
9
```

### 🔹 10. count() – Count Occurrences

```
text = "Python is easy, Python is powerful"

print(text.count("Python"))
```

🎯 Output:

```
2
```

## 💡 String Immutability Example

```
word = "Hello"
```

# word[0] = 'Y' ❌ (Error)

word = "Y" + word[1:]  # ✅ Create new string

print(word)

🎯 Output:

Yello

✅ Strings cannot be changed directly — you must create a new one.

## 🧮 Summary Table

| Operation | Description | Example | Output |
|---|---|---|---|
| len() | Length of string | len("Python") | 6 |
| + | Concatenate | "Hi" + " Naveen" | Hi Naveen |
| * | Repeat | "Hi" * 3 | HiHiHi |
| in | Membership test | 'Py' in "Python" | True |
| upper() | Uppercase | "hi".upper() | HI |
| lower() | Lowercase | "HI".lower() | hi |
| strip() | Remove spaces | " hi ".strip() | hi |
| replace() | Replace text | "Hi".replace("i","ello") | Hello |
| split() | Split string | "a,b,c".split(",") | ['a','b','c'] |
| join() | Join list | " ".join(['a','b']) | a b |

## 🧠 Real-Life Example: Cleaning User Input

user_input = "  naveen  "

clean_name = user_input.strip().capitalize()

```
print("Welcome,", clean_name)
```

🎯 Output:

```
Welcome, Naveen
```

## 🏁 Conclusion

By mastering **strings and their operations**, you can:
✅ Work with text data effectively
✅ Clean and preprocess user input
✅ Build features like searching, formatting, and text manipulation

# Day 31: String Methods

## 🧭 Day 31: String Methods in Python

### 🎯 Goal:

To learn and understand all the **important built-in string methods** in Python, which help you perform common text-related operations like formatting, searching, validation, and modification easily.

## 🧠 What are String Methods?

String methods are **built-in functions** that perform specific tasks on string objects.

✅ They **don't modify** the original string (because strings are immutable).
✅ Instead, they **return a new string** or a **boolean value**.

## 🧩 Categories of String Methods

We can divide string methods into five major groups:

1. **Case Conversion Methods**
2. **Searching and Replacing Methods**
3. **Validation Methods**
4. **Splitting and Joining Methods**
5. **Formatting and Utility Methods**

Let's understand each one with syntax, examples, and outputs 👇

## 🧮 Case Conversion Methods

These methods help you convert the case (uppercase/lowercase/title case) of strings.

### 🔹 upper()

Converts all characters in a string to **uppercase**.

```
text = "python programming"

print(text.upper())
```

🎯 Output:

```
PYTHON PROGRAMMING
```

### 🔹 lower()

Converts all characters in a string to **lowercase**.

```
text = "HELLO WORLD"

print(text.lower())
```

🎯 Output:

```
hello world
```

### 🔹 title()

Converts the **first letter** of each word to uppercase.

```
text = "learn python programming"

print(text.title())
```

🎯 Output:

```
Learn Python Programming
```

### 🔹 capitalize()

Converts the **first letter** of the string to uppercase and the rest to lowercase.

```
text = "python is fun"

print(text.capitalize())
```

🎯 Output:

```
Python is fun
```

### 🔹 swapcase()

Converts **uppercase to lowercase** and **lowercase to uppercase**.

text = "PyThOn"

print(text.swapcase())

🎯 Output:

pYtHoN

## 🔍 Searching and Replacing Methods

These help you **find** or **replace** text within strings.

### 🔹 find(substring, start, end)

Finds the **first occurrence** of a substring. Returns -1 if not found.

text = "Python is amazing"

print(text.find("is"))

print(text.find("Java"))

🎯 Output:

7

-1

### 🔹 rfind(substring)

Finds the **last occurrence** of a substring.

text = "Python is easy, Python is powerful"

print(text.rfind("Python"))

🎯 Output:

17

### 🔹 index(substring)

Similar to find(), but raises an **error** if the substring is not found.

text = "Learn Python"

print(text.index("Python"))

# print(text.index("Java"))  # ❌ Error

🎯 Output:

6

### 🔹 replace(old, new, count)

Replaces occurrences of a substring with another substring.

text = "I love Java"

print(text.replace("Java", "Python"))

🎯 Output:

I love Python

## 🧾 Validation Methods

These methods **check the content** of a string and return **True or False**.

### 🔹 isalnum()

Returns True if all characters are **letters or numbers**.

text = "Python123"

print(text.isalnum())

🎯 Output:

True

### 🔹 isalpha()

Returns True if all characters are **letters only**.

text = "Python"

print(text.isalpha())

🎯 Output:

True

### 🔹 isdigit()

Returns True if all characters are **digits**.

```
num = "12345"

print(num.isdigit())
```

🎯 Output:

```
True
```

### 🔹 islower()

Returns True if all letters are **lowercase**.

```
text = "python"

print(text.islower())
```

🎯 Output:

```
True
```

### 🔹 isupper()

Returns True if all letters are **uppercase**.

```
text = "HELLO"

print(text.isupper())
```

🎯 Output:

```
True
```

### 🔹 isspace()

Returns True if the string contains **only whitespace**.

```
text = "   "

print(text.isspace())
```

🎯 Output:

```
True
```

### 🔹 istitle()

Returns True if each word in the string is **title-cased**.

```
text = "Python Programming"

print(text.istitle())
```

🎯 Output:

True

## ✂️ Splitting and Joining Methods

### 🔹 split(separator, maxsplit)

Splits the string into a list using the specified **separator**.

```
text = "Python,SQL,PowerBI"

print(text.split(","))
```

🎯 Output:

['Python', 'SQL', 'PowerBI']

### 🔹 rsplit(separator, maxsplit)

Splits from the **right side**.

```
text = "a,b,c,d"

print(text.rsplit(",", 1))
```

🎯 Output:

['a,b,c', 'd']

### 🔹 join(iterable)

Joins the elements of an iterable (like list or tuple) into a string.

```
words = ['Python', 'is', 'fun']

print(" ".join(words))
```

🎯 Output:

Python is fun

### 🔹 partition(separator)

Splits the string into **three parts** — before, separator, after.

```
text = "email@example.com"

print(text.partition("@"))
```

🎯 Output:

('email', '@', 'example.com')

### 🔹 rpartition(separator)

Similar to partition(), but starts splitting from the **right**.

```
path = "C:/Users/Naveen/Desktop"

print(path.rpartition("/"))
```

🎯 Output:

('C:/Users/Naveen', '/', 'Desktop')

## 🧮 Formatting and Utility Methods

### 🔹 startswith(prefix)

Checks if a string **starts** with a given substring.

```
text = "Python Programming"

print(text.startswith("Python"))
```

🎯 Output:

True

### 🔹 endswith(suffix)

Checks if a string **ends** with a given substring.

```
text = "data.csv"

print(text.endswith(".csv"))
```

🎯 Output:

True

### 🔹 count(substring)

Returns the number of times a substring appears in the string.

```
text = "Python Python Python"

print(text.count("Python"))
```

🎯 Output:

3

### 🔹 center(width, fillchar)

Centers the string in a field of specified width.

```
text = "Python"

print(text.center(15, "-"))
```

🎯 Output:

----Python----

### 🔹 ljust(width, fillchar)

Left aligns the string.

```
print("Python".ljust(10, "*"))
```

🎯 Output:

Python****

### 🔹 rjust(width, fillchar)

Right aligns the string.

```
print("Python".rjust(10, "*"))
```

🎯 Output:

****Python

### 🔹 zfill(width)

Pads the string with zeros on the left.

```
num = "42"

print(num.zfill(5))
```

🎯 Output:

00042

### 🔹 format()

Used to insert values into a string.

```
name = "Naveen"

age = 21

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

🎯 Output:

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

### 🔹 f-Strings (Modern String Formatting)

Introduced in Python 3.6 — a simpler and faster way to format strings.

```
name = "Naveen"

course = "Python"

print(f"Hello, {name}! Welcome to {course} class.")
```

🎯 Output:

```
Hello, Naveen! Welcome to Python class.
```

## 🟦 Summary Table

| Method | Description | Example | Output |
|---|---|---|---|
| upper() | Uppercase | 'hi'.upper() | 'HI' |
| lower() | Lowercase | 'HI'.lower() | 'hi' |
| title() | Title case | 'python rocks'.title() | 'Python Rocks' |
| find() | Find substring | 'hi python'.find('py') | 3 |
| replace() | Replace text | 'hi'.replace('hi','hey') | 'hey' |
| startswith() | Check prefix | 'data.csv'.startswith ('data') | True |
| endswith() | Check suffix | 'data.csv'.endswith( '.csv') | True |
| count() | Count occurrences | 'hi hi hi'.count('hi') | 3 |
| split() | Split string | 'a,b'.split(',') | ['a','b'] |
| join() | Join list | '-'.join(['a','b']) | 'a-b' |
| format() | Format string | '{} {}'.format('Hi','Navee n') | 'Hi Naveen' |
| isalnum() | Check alphanumeric | 'Python3'.isalnum() | True |
| isalpha() | Check letters | 'Python'.isalpha() | True |

## 🧠 Real-Life Example: Email Validation

email = "naveen@example.com"

if email.endswith("@example.com"):

```
    print("✅ Valid company email")
else:
    print("❌ Invalid email")
```

🎯 Output:

✅ Valid company email

## 🏁 Conclusion

By mastering **String Methods**, you can:
✅ Clean and preprocess text data
✅ Validate user input efficiently
✅ Format output beautifully
✅ Build text-based applications like chatbots, input forms, etc.

# Day 32: String Formatting

✅ **Day 32: String Formatting in Python**

### 🎯 Objective:

Learn how to **format strings** in Python to display data in a clean and readable way. You'll understand how to use the **format() method**, **f-strings**, and **percent formatting** for dynamic text.

## 🔹 1. What is String Formatting?

String formatting means **inserting variables or values inside a string** in a readable and structured manner.

👉 Example without formatting:

```
name = "Naveen"
age = 21
print("My name is " + name + " and I am " + str(age) + " years old.")
```

❌ Problem: It's messy and hard to read — plus, you must convert numbers to strings manually.

✅ Solution: Use **String Formatting** to make it cleaner and more powerful.

## 🔹 2. Different Ways to Format Strings in Python

Python offers **3 main ways** to format strings:

1. **Using format() method**
2. **Using f-strings (formatted string literals)**
3. **Using % (percent formatting)**

Let's learn them one by one 👇

## 🧩 Using format() Method

### ✅ Syntax:

"{}".format(value)

You can insert multiple values:

"{} is {} years old.".format("Naveen", 21)

### ✅ Output:

Naveen is 21 years old.

### 🔸 Positional Formatting

You can specify the position of arguments:

"{1} is learning {0}".format("Python", "Naveen")

### ✅ Output:

Naveen is learning Python

Here {1} uses the second argument "Naveen", and {0} uses the first "Python".

### 🔸 Named Placeholders

You can name the variables:

"{name} is from {country}".format(name="Naveen", country="India")

### ✅ Output:

Naveen is from India

### 🔸 Formatting Numbers

You can format decimal numbers:

num = 3.1415926

print("Value: {:.2f}".format(num))

### ✅ Output:

Value: 3.14

(.2f means 2 decimal places)

## 🧩 Using f-Strings (Python 3.6+)

**f-Strings** are the **modern and fastest** way to format strings.

### ✅ Syntax:

f"some text {variable}"

Example:

name = "Naveen"

age = 21

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

### ✅ Output:

My name is Naveen and I am 21 years old.

### 🔸 You can even perform calculations inside f-strings:

a = 10

b = 5

print(f"Sum of {a} and {b} is {a + b}")

### ✅ Output:

Sum of 10 and 5 is 15

### 🔸 Formatting numbers in f-strings:

price = 49.9567

print(f"Price: {price:.2f}")

✅ **Output:**

Price: 49.96

🔸 **Date formatting example:**

from datetime import datetime

today = datetime.now()

print(f"Today is {today:%A, %B %d, %Y}")

✅ **Output:**

Today is Sunday, November 02, 2025

## 🧩 Using Percent (%) Formatting

This is an **older style** but still used sometimes.

✅ **Syntax:**

"%type" % value

Example:

name = "Naveen"

age = 21

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

✅ **Output:**

My name is Naveen and I am 21 years old.

🔸 **Common Format Codes:**

| Code | Meaning | Example |
|---|---|---|
| %s | String | "Name: %s" % "Naveen" |
| %d | Integer | "Age: %d" % 21 |
| %f | Float | "Price: %.2f" % 25.4567 |

## 🧠 Comparison Between Formatting Methods

| Method | Description | Example |
|---|---|---|
| format() | Flexible and powerful | "{} {}".format(a, b) |
| f-string | Modern, fastest, cleanest | f"{a} {b}" |
| % formatting | Old and less readable | "%s %d" % (a, b) |

## ⚡ Real-Life Example

Let's use string formatting in a **bill receipt generator** 👇

item = "Laptop"

price = 59999.99

quantity = 2

total = price * quantity

print(f"Item: {item}")

print(f"Price: ₹{price:.2f}")

print(f"Quantity: {quantity}")

print(f"Total: ₹{total:.2f}")

### ✅ Output:

Item: Laptop

Price: ₹59999.99

Quantity: 2

Total: ₹119999.98

## 💡 Why String Formatting Is Important

✅ Makes output **clean and user-friendly**
✅ Easy to **combine text and variables**
✅ Helps in **report generation, logging, and dashboards**
✅ **f-Strings** are the most **efficient** and **modern** choice

## 🎯 Summary

| Concept | Description |
|---|---|
| format() | Uses {} placeholders for inserting values |
| f-strings | Best and modern way, introduced in Python 3.6 |
| % formatting | Old method, less used now |
| Formatting Numbers | Use .2f, .3f, etc. for decimals |
| f-strings power | Can do inline calculations and formatting |

## 🧠 Quick Practice

name = "Naveen"

marks = 95.6789

# Try all three formatting methods

print("1. Using format(): {} scored {:.2f}%".format(name, marks))

print(f"2. Using f-string: {name} scored {marks:.2f}%")

print("3. Using %% formatting: %s scored %.2f%%" % (name, marks))

# Day 33: Functions - Defining and Calling

## ✅ Day 33: Functions – Defining and Calling in Python

### 🎯 Objective:

Learn what **functions** are, why they are important, how to **define and call functions** in Python, and how they make your code **modular, reusable, and organized**.

## 🔹 1. What is a Function?

A **function** is a **block of reusable code** that performs a specific task.
You can **define it once** and **use (call) it multiple times**.

👉 Think of a function as a **machine**:
You give it an **input**, it does some **work**, and gives an **output**.

### 🧩 Real-Life Analogy

Imagine a **coffee machine** ☕:

- Input → Water + Coffee beans
- Machine (function) → Makes coffee
- Output → Cup of Coffee

Similarly, a Python function:

- Input → arguments
- Function → performs logic
- Output → result

## 🔹 2. Why Use Functions?

✅ **Avoid Code Repetition** — write once, use many times
✅ **Easier to Debug** — errors are isolated
✅ **Modular Code** — organize logic into small parts
✅ **Reusability** — use the same function in multiple programs

## 🔹 3. Function Syntax in Python

### ✅ Basic Syntax:

```
def function_name():

    # function body

    # (statements)
```

- def → keyword used to define a function
- function_name → name you choose for the function
- () → parentheses (may contain parameters)
- : → indicates the start of the function body

- Inside → indented code (the body)

## 🔹 4. Defining and Calling a Function

Example : Simple Function

```
def greet():
    print("Hello, welcome to Python!")
```

Now, to **execute (call)** the function:

```
greet()
```

### ✅ Output:

```
Hello, welcome to Python!
```

👉 The function greet() runs only when **called**.

Example : Function Called Multiple Times

```
def greet():
    print("Good Morning!")


greet()
greet()
```

### ✅ Output:

```
Good Morning!
Good Morning!
```

✔ The same function can be reused as many times as you want.

## 🔹 5. Function with Parameters

You can pass **data** into a function using **parameters** (also called arguments).

Example:

```
def greet(name):
    print(f"Hello, {name}! Welcome to Python.")
```

Calling the function:

```
greet("Naveen")

greet("Ravi")
```

✅ **Output:**

```
Hello, Naveen! Welcome to Python.

Hello, Ravi! Welcome to Python.
```

🔸 **Multiple Parameters Example**

```
def add_numbers(a, b):
    print("Sum:", a + b)
```

Calling the function:

```
add_numbers(10, 20)
```

✅ **Output:**

```
Sum: 30
```

## 🔹 6. Function Returning a Value

Functions can **return** results using the **return** keyword.

Example:

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


result = add(5, 10)

print("Result:", result)
```

✅ **Output:**

```
Result: 15
```

✅ The return keyword sends data **back to where the function was called**.

🔸 **Why Use Return?**

Because sometimes you need to **store** or **use** the result later, not just print it.

Example:

```
def square(num):
    return num * num


x = square(4)
print("Square:", x)
```

✅ **Output:**

```
Square: 16
```

## 🔹 7. Default Parameters

If you don't provide a value for a parameter, Python uses a **default value**.

### Example:

```
def greet(name="Guest"):
    print(f"Hello, {name}!")
```

### Calls:

```
greet("Naveen")
greet()
```

✅ **Output:**

```
Hello, Naveen!
Hello, Guest!
```

## 🔹 8. Function with Multiple Return Values

You can return **more than one value** from a function using a tuple.

### Example:

```
def calculator(a, b):
    return a + b, a - b, a * b
```

```
add, sub, mul = calculator(10, 5)

print("Addition:", add)

print("Subtraction:", sub)

print("Multiplication:", mul)
```

✅ **Output:**

```
Addition: 15

Subtraction: 5

Multiplication: 50
```

## 🔹 9. Local and Global Variables

- **Local Variable** → defined **inside** a function (accessible only there)
- **Global Variable** → defined **outside** a function (accessible everywhere)

Example:

```
x = 10  # Global variable


def display():

    y = 5  # Local variable

    print("Inside function:", x + y)


display()

print("Outside function:", x)
```

✅ **Output:**

```
Inside function: 15

Outside function: 10
```

## 🔹 10. Nested Function (Function Inside Function)

Python allows defining one function **inside another**.

Example:

```
def outer():
    print("Outer function")

    def inner():
        print("Inner function")
    inner()

outer()
```

✅ **Output:**

Outer function

Inner function

## 🔹 11. Real-Life Example – Billing Function

```
def generate_bill(name, quantity, price_per_item):
    total = quantity * price_per_item
    print(f"Customer: {name}")
    print(f"Total Bill: ₹{total:.2f}")
    return total

# Calling
generate_bill("Naveen", 3, 120)
```

✅ **Output:**

Customer: Naveen

Total Bill: ₹360.00

## 💡 12. Key Points to Remember

| Concept | Description |
|---|---|
| def | Used to define a function |
| Parameters | Data passed to the function |
| return | Sends a value back |
| Local Variable | Exists only inside a function |
| Global Variable | Accessible anywhere |
| Reusability | You can call the same function many times |

## ⚙️ 13. Practice Questions

Define a function square() that returns the square of a number.
Create a function area_of_circle(radius) that returns the area using πr².
Write a function is_even(num) that returns True if even, else False.
Create a function greet_user(name, age) that prints “Hello {name}, you are {age} years old.”

## 🧠 14. Summary

| Topic | Description | Example |
|---|---|---|
| Function Definition | Create a reusable block of code | def greet(): |
| Function Call | Executes the function | greet() |
| Parameters | Accept inputs | def add(a,b): |
| Return | Send result back | return a+b |
| Default Value | Predefined argument | def greet(name="Guest") |
| Nested Function | Function inside another | def outer(): def inner(): |

## ⬅️ END Conclusion

Functions are the **building blocks** of Python programming.
They help you **organize**, **reuse**, and **simplify** your code.
Mastering functions is a key step toward becoming a **professional Python developer**. 🐍💪

# Day 34: Function Arguments and Return Values

### ✅ Day 34: Function Arguments and Return Values in Python

### 🎯 Objective:

Today you'll learn about **different types of function arguments** in Python — how to pass values to functions, use default values, handle variable numbers of arguments, and how **return values** work in detail.

By the end of this lesson, you'll understand how to **create flexible and powerful functions** that can take inputs and send outputs effectively. 🚀

## 🔹 1. Recap: What Is a Function?

A **function** is a block of reusable code that performs a specific task.
It can take **inputs (arguments)** and can also **return outputs (values)**.

👉 Example (simple recall):

```
def greet():
    print("Hello, Naveen!")
```

Here:

- greet → Function name
- () → No arguments
- Inside the function → Task to perform
- print() → Output is displayed directly, not returned

## 🔹 2. What Are Function Arguments?

**Arguments (or parameters)** are the values we pass **inside the parentheses** when calling a function.

👉 Example:

```
def greet(name):
    print("Hello", name)
```

Here:

- name → Parameter (defined in function)
- When you call greet("Naveen"), the value "Naveen" → Argument (actual value passed)

🟦 **Parameter** = variable in function definition
🟦 **Argument** = actual value you pass to it

## 🔷 3. Types of Function Arguments in Python

Python supports **five main types** of arguments:

| Type | Description |
|---|---|
| | Positional Arguments |
| | Keyword Arguments |
| | Default Arguments |
| | Variable-Length Arguments (*args) |
| | Keyword Variable-Length Arguments (**kwargs) |

Let's explore each in detail 👇

## 🧩 Positional Arguments

Arguments are passed **in the same order** as parameters are defined.

Example:

```
def student_info(name, age):
    print("Name:", name)
    print("Age:", age)
```

student_info("Naveen", 21)

✅ **Output:**

Name: Naveen

Age: 21

If you change the order:

student_info(21, "Naveen")

❌ **Output:**

Name: 21

Age: Naveen

👉 Hence, order matters for positional arguments.

## 🧩 Keyword Arguments

You can specify **parameter names** while calling a function — order doesn't matter.

Example:

```
def student_info(name, age):
    print("Name:", name)
    print("Age:", age)


student_info(age=21, name="Naveen")
```

✅ **Output:**

Name: Naveen

Age: 21

✔ Keyword arguments improve **readability** and **avoid confusion**.

## 🧩 Default Arguments

You can give **default values** to parameters.
If no value is passed, Python uses the default.

Example:

```
def greet(name="Guest"):
    print("Hello", name)


greet("Naveen")
greet()
```

✅ **Output:**

```
Hello Naveen
Hello Guest
```

💡 Default arguments make parameters **optional**.

## 🧩 Variable-Length Arguments – *args

If you don't know how many arguments will be passed, use *args.
It collects all extra **positional arguments** as a **tuple**.

Example:

```
def add_numbers(*nums):
    print(nums)
    total = sum(nums)
    print("Sum:", total)


add_numbers(2, 4, 6)
add_numbers(10, 20, 30, 40)
```

✅ **Output:**

```
(2, 4, 6)
Sum: 12
(10, 20, 30, 40)
Sum: 100
```

💡 Use *args when you want **flexibility** in number of inputs.

## 🧩 Keyword Variable-Length Arguments – **kwargs

If you want to accept multiple **key-value pairs** (like a dictionary), use **kwargs.

### Example:

```
def display_info(**details):
    for key, value in details.items():
        print(f"{key}: {value}")


display_info(name="Naveen", age=21, country="India")
```

### ✅ Output:

name: Naveen

age: 21

country: India

💡 **kwargs is often used when parameters are **optional** or **dynamic** (like configuration or user details).

## 🔷 4. Combining All Argument Types

You can combine different types of arguments — but **order matters**.

### Correct Order:

Positional → Default → *args → **kwargs

### Example:

```
def demo(a, b=10, *args, **kwargs):
    print("a:", a)
    print("b:", b)
    print("args:", args)
    print("kwargs:", kwargs)
```

```
demo(5, 20, 30, 40, x=100, y=200)
```

✅ **Output:**

```
a: 5
b: 20
args: (30, 40)
kwargs: {'x': 100, 'y': 200}
```

## 🔹 5. Return Values in Functions

The return statement sends a **result** back from a function to the caller.

✅ **Syntax:**

```
def function_name():
    return value
```

Example : Returning a single value

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


result = add(10, 5)
print("Sum:", result)
```

✅ **Output:**

```
Sum: 15
```

Example : Returning multiple values

You can return multiple values as a **tuple**.

```
def calculate(a, b):
    return a + b, a - b, a * b


add, sub, mul = calculate(10, 5)
```

```
print("Addition:", add)

print("Subtraction:", sub)

print("Multiplication:", mul)
```

✅ **Output:**

```
Addition: 15

Subtraction: 5

Multiplication: 50
```

Example : Using return to stop a function early

```
def check_number(num):
    if num < 0:
        return "Negative"
    return "Positive"


print(check_number(-5))

print(check_number(7))
```

✅ **Output:**

```
Negative

Positive
```

💡 Once return is executed, the function **exits immediately**.

## 🔹 6. Real-Life Example: Student Marks Calculation

```
def student_result(name, *marks):
    avg = sum(marks) / len(marks)
    if avg >= 90:
        grade = "A+"
    elif avg >= 75:
```

```
    grade = "A"
  else:
    grade = "B"

  return name, avg, grade

# Calling
student_name, average, grade = student_result("Naveen", 95, 88, 90, 92)
print(f"Student: {student_name}, Average: {average:.2f}, Grade: {grade}")
```

✅ **Output:**

```
Student: Naveen, Average: 91.25, Grade: A+
```

## 🧠 7. Important Points

| Concept | Description |
|---|---|
| Positional Argument | Values passed in correct order |
| Keyword Argument | Specify by name; order doesn't matter |
| Default Argument | Used when no value is passed |
| *args | Multiple positional arguments |
| **kwargs | Multiple keyword arguments |
| return | Sends result back to caller |
| Multiple Return | Returns tuple values |

## ⚙️ 8. Practice Questions

Create a function multiply(a, b=1) that multiplies two numbers (second is optional).
Write a function student_details(**info) that prints student details dynamically.
Make a function calc_total(*prices) that returns the sum of all prices.
Write a function stats(a, b) that returns sum, average, and product of two numbers.

## 💡 9. Summary

| Type | Example | Description |
|---|---|---|
| Positional | add(5, 10) | Order matters |
| Keyword | add(b=10, a=5) | Order doesn't matter |
| Default | add(a, b=10) | Has a fallback value |
| *args | add(1, 2, 3) | Multiple positional |
| **kwargs | add(x=1, y=2) | Multiple keyword |
| return | return result | Send output from function |

# Day 35: Lambda Functions

✅ **Day 35: Lambda Functions (Anonymous Functions) in Python**

## 🎯 Objective:

Learn what **lambda functions** (also called **anonymous functions**) are, how they differ from regular functions, their **syntax**, **use cases**, and **real-world examples** in data manipulation, filtering, and sorting.

By the end of this lesson, you'll be able to use lambda functions to **write cleaner, shorter, and faster** Python code. 🚀

## 🔹 1. What is a Lambda Function?

A **lambda function** is a **small, one-line anonymous function** — meaning it **doesn't have a name**.

It is used for **short, simple operations** where defining a full function with def would be unnecessary.

### ✅ Syntax:

lambda arguments: expression

- lambda → keyword used to define the function
- arguments → input parameters (like normal function parameters)
- expression → single expression (no statements, no loops)

⚡ It automatically returns the result of that expression — **no need for return keyword!**

## 🧩 2. Example 1: Simple Lambda Function

Using def:

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


print(add(5, 3))
```

Using lambda:

```
add = lambda a, b: a + b

print(add(5, 3))
```

### ✅ Output:

8

👉 Both give the same output, but the lambda version is shorter and cleaner.

## 🧩 3. Example 2: Lambda Function with One Argument

```
square = lambda x: x * x

print(square(4))
```

### ✅ Output:

16

## 🧩 4. Example 3: Lambda Function with No Arguments

```
greet = lambda: print("Hello, Naveen!")

greet()
```

✅ **Output:**

```
Hello, Naveen!
```

## 🔹 5. Why Use Lambda Functions?

Lambda functions are used when:
✅ The function is **very short**
✅ You need a function **only once** (temporary)
✅ You want to use it **inside another function**

Commonly used with built-in functions like:

- map()
- filter()
- reduce()
- sorted()

## 🔹 6. Using Lambda with map()

The **map()** function applies a given function to **each element of a list (or iterable)**.

Example:

```
numbers = [1, 2, 3, 4, 5]

squares = list(map(lambda x: x * x, numbers))

print(squares)
```

✅ **Output:**

```
[1, 4, 9, 16, 25]
```

💡 map() applies the lambda to each element — perfect for transformations.

## 🔹 7. Using Lambda with filter()

The **filter()** function selects elements based on a condition (True/False).

Example:

```
numbers = [10, 25, 30, 45, 50]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
```

```
print(even_numbers)
```

✅ **Output:**

```
[10, 30, 50]
```

💡 filter() keeps only numbers that satisfy the condition.

## 🔹 8. Using Lambda with reduce()

The **reduce()** function (from functools module) applies a function cumulatively to elements.

Example:

```
from functools import reduce


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

sum_all = reduce(lambda x, y: x + y, numbers)

print(sum_all)
```

✅ **Output:**

```
15
```

💡 It works like: (((1+2)+3)+4)+5

## 🔹 9. Using Lambda with sorted()

You can use a lambda as a **key function** to sort lists of tuples or dictionaries.

Example : Sorting by second element

```
data = [(1, 50), (2, 20), (3, 70)]

sorted_data = sorted(data, key=lambda x: x[1])

print(sorted_data)
```

✅ **Output:**

```
[(2, 20), (1, 50), (3, 70)]
```

Example : Sorting names by length

```
names = ["Naveen", "Raj", "Krishna", "Teja"]
```

```
sorted_names = sorted(names, key=lambda name: len(name))

print(sorted_names)
```

✅ **Output:**

```
['Raj', 'Teja', 'Naveen', 'Krishna']
```

## 🔹 10. Lambda with Conditional Expression

You can use **if-else** inside lambda functions (for short conditions).

Example :

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

print(check_even(7))
```

✅ **Output:**

```
Odd
```

Example :

```
grade = lambda marks: "Pass" if marks >= 40 else "Fail"

print(grade(85))

print(grade(30))
```

✅ **Output:**

```
Pass

Fail
```

## 🔹 11. Real-Life Example — Price Discount Calculator

```
discount = lambda price: price * 0.9 if price > 1000 else price

prices = [500, 1200, 700, 2000]

new_prices = list(map(discount, prices))

print(new_prices)
```

✅ **Output:**

```
[500, 1080.0, 700, 1800.0]
```

💡 Items priced above ₹1000 get a 10% discount automatically.

## 🔹 12. Lambda vs Regular Function

| Feature | Regular Function | Lambda Function |
|---|---|---|
| Defined with | def | lambda |
| Has a name | Yes | Usually no |
| Can have multiple statements | Yes | No (only one expression) |
| Has return | Must use return | Implicit return |
| Used for | Complex logic | Small, quick tasks |

## 🔹 13. Limitations of Lambda Functions

⚠️ **Lambda functions are limited** —
They can contain **only one expression**, no loops, and no multiple statements.

❌ Not allowed:

```
lambda x: for i in range(x): print(i)  # ❌ Invalid
```

✅ Correct:

```
lambda x: x * 2  # ✅ Single expression only
```

## 💡 14. Practice Questions

Try these yourself 👇

Create a lambda function to find the cube of a number.
Use filter() and lambda to get all numbers greater than 50 from a list.
Use map() and lambda to convert a list of temperatures from Celsius to Fahrenheit.
Use reduce() and lambda to find the product of all numbers in a list.
Use sorted() with lambda to sort a list of students by their marks.

## 🧠 15. Summary

| Concept | Example | Description |
|---|---|---|
| Basic Syntax | lambda x: x + 10 | Defines a one-line function |
| With Arguments | lambda a, b: a + b | Multiple inputs |
| With map() | map(lambda x: x*x, list) | Apply to each element |
| With filter() | filter(lambda x: x>10, list) | Keep matching items |
| With reduce() | reduce(lambda x,y: x+y, list) | Combine all elements |
| Conditional | lambda x: 'Even' if x%2==0 else 'Odd' | Inline condition |

### 🔚 Conclusion

Lambda functions are **powerful, compact tools** that make your code elegant and expressive.
They shine when you want to **quickly define small logic** — especially inside functions like map(), filter(), and reduce().

Mastering lambda functions will make your Python code **more functional, cleaner, and faster to write.** 🐍

## Day 36: Scope of Variables (Local and Global)

### ✅ Day 36: Scope of Variables (Local and Global Variables) in Python

### 🎯 Objective:

Understand what **variable scope** means — how and where variables can be accessed in a Python program.
You'll learn about **local**, **global**, **nonlocal**, and **built-in** scopes with examples, diagrams, and real-life use cases.

By the end of this lesson, you'll clearly know **which variable Python will use at which point** in your code. 🚀

## 🔹 1. What is Variable Scope?

**Scope** refers to the **area of a program** where a variable is **accessible or visible**.

👉 Every variable in Python has a **scope** that decides:

- **Where** the variable can be used
- **How long** it exists (its lifetime)

🧠 **Example 1:**

```
x = 10  # global variable


def show():
    y = 5  # local variable
    print("Inside function:", y)


show()
print("Outside function:", x)
```

✅ **Output:**

```
Inside function: 5
Outside function: 10
```

🟦 Here:

- x is **global** → accessible anywhere
- y is **local** → accessible only inside show()

## 🔹 2. Types of Variable Scopes in Python

Python follows a rule called **LEGB Rule** 🔍

| Level | Scope Type | Description |
|---|---|---|
| **L** | **Local** | Inside the current function |
| **E** | **Enclosing** | Inside nested (outer) functions |
| **G** | **Global** | Declared at the top level of the script |
| **B** | **Built-in** | Predefined names in Python (e.g. len, print) |

## 🧩 3. Local Variables

A **local variable** is defined **inside a function** and can only be used **within that function**.

Example:

```
def greet():
    name = "Naveen"  # local variable
    print("Hello", name)

greet()
print(name)  # ❌ Error: name is not defined
```

✅ **Output:**

```
Hello Naveen
```

Traceback (most recent call last):

...

NameError: name 'name' is not defined

💡 **Reason:**
name exists **only inside** greet(). Outside it, Python can't find it.

## 🧩 4. Global Variables

A **global variable** is declared **outside any function**, and it's accessible **everywhere** in the code — both inside and outside functions.

Example:

```
language = "Python"  # global variable


def show_language():
    print("I love", language)


show_language()
print("Outside function:", language)
```

✅ **Output:**

I love Python

Outside function: Python

## ⚠️ 5. Modifying a Global Variable Inside a Function

If you **try to modify** a global variable inside a function **without declaring it as global**, Python will **create a new local variable** instead of changing the global one.

Example (without global keyword):

```
count = 10

def increase():
    count = count + 1  # ❌ Error
    print("Inside:", count)

increase()
```

❌ **Output:**

```
UnboundLocalError: local variable 'count' referenced before assignment
```

💡 Python assumes count is a **local variable**, but since it's being modified before defined, it throws an error.

## ✅ 6. Correct Way — Using global Keyword

Use the **global** keyword to tell Python you're working with the **global variable**.

Example:

```
count = 10

def increase():
    global count
    count = count + 1
    print("Inside function:", count)

increase()
```

```
print("Outside function:", count)
```

✅ **Output:**

Inside function: 11

Outside function: 11

💡 Now the global variable is modified correctly.

## 🧩 7. Enclosing Scope (Nested Functions)

If you define a **function inside another function**, the **inner function** can access variables from the **outer (enclosing) function**.

Example:

```
def outer():
    x = "outer variable"

    def inner():
        print("Accessing from inner:", x)

    inner()

outer()
```

✅ **Output:**

Accessing from inner: outer variable

💡 The inner function can access the **enclosing function's variable**.

## 🔹 8. Modifying Enclosing Variables — Using nonlocal

To modify an enclosing (non-global) variable, use the **nonlocal** keyword.

Example:

```
def outer():
    x = 5

    def inner():
        nonlocal x
        x = x + 1
        print("Inside inner:", x)

    inner()
    print("Outside inner (in outer):", x)

outer()
```

✅ **Output:**

Inside inner: 6

Outside inner (in outer): 6

💡 nonlocal allows the inner function to modify a variable from the **enclosing scope** (not global).

## 🧩 9. Built-in Scope

This is the **outermost** scope — names that are **predefined in Python**, like:
print(), len(), max(), range(), etc.

Example:

```
print(len("Naveen"))  # 'len' is from built-in scope
```

You can view all built-in names:

```
import builtins

print(dir(builtins))
```

## 🧩 10. LEGB Rule Summary (Order of Variable Resolution)

When Python encounters a variable, it searches in this order:

**Local** → Inside current function
**Enclosing** → Inside outer function (if nested)
**Global** → At the top level of the program
**Built-in** → Predefined Python names

Example:

```
x = "global"


def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print(x)
    inner()


outer()
```

✅ **Output:**

```
local
```

💡 Python found x in the **local** scope first and didn't need to look further.

## 🧩 11. Real-Life Example — Bank Account Balance

```
balance = 1000  # global variable


def transaction():
    global balance
    deposit = 500  # local variable
    balance += deposit
    print("After deposit:", balance)


transaction()
print("Final balance:", balance)
```

✅ **Output:**

```
After deposit: 1500
Final balance: 1500
```

💡 Global balance is modified across the program.

## 🧩 12. Variable Lifetime

- A **local variable** is created when the function starts and **destroyed** when it ends.
- A **global variable** exists until the program finishes.

Example:

```
def demo():
    name = "Naveen"
```

```
    print(name)


demo()

print(name)  # ❌ Error
```

💡 name is destroyed after demo() ends.

## 🧠 13. Practice Questions

Try these yourself 👇

Define a global variable and modify it inside a function using the global keyword.

Create a nested function and access the outer function variable using nonlocal.

Show what happens if you try to modify a global variable inside a function without global.

Write an example to demonstrate the **LEGB** rule.

Use local and global variables to simulate a login counter.

## 🧩 14. Summary Table

| Type | Declared Where | Accessible Where | Modified Using | Lifetime |
|---|---|---|---|---|
| **Local** | Inside a function | Only in that function | N/A | Function runs |
| **Enclosing** | In outer function | Inner functions | nonlocal | Outer function |
| **Global** | Outside all functions | Everywhere | global | Entire program |
| **Built-in** | Python library | Everywhere | Not modifiable | Python runtime |

### 🔚 Conclusion

Understanding variable **scope** is **crucial** to writing error-free code.
It prevents unexpected bugs like variable overwriting or incorrect data access.

Remember the **LEGB Rule** —

Python always looks for a variable from **Local → Enclosing → Global → Built-in** scope in that order.

By mastering scopes, you'll control **which data lives where** — a must-have skill for all serious Python developers. 🧠🐍

## Day 37: Recursion in Python

### ✅ Day 37: Recursion in Python — Complete Detailed Explanation

### 🎯 Objective:

Understand what **recursion** is, how it works in Python, why it's powerful, and how to use it efficiently to solve problems like factorial, Fibonacci series, and sum of numbers.
By the end, you'll understand **how functions can call themselves** and how to control that process using the **base condition**. 🧠🐍

## 🔹 1. What is Recursion?

**Recursion** is a process where **a function calls itself** in order to solve a problem.

It is based on the idea of **breaking a big problem into smaller, similar sub-problems** until you reach a condition that can be solved directly (called the **base case**).

### 🧩 Example (Simple Example):

```
def greet():
    print("Hello Naveen!")
    greet()  # Function calling itself


greet()
```

⚠️ This will **never stop** — it keeps calling itself again and again until **Python runs out of memory**.
You'll get an error:

RecursionError: maximum recursion depth exceeded

💡 **Lesson:** Every recursive function must have a **base condition** to stop recursion.

## 🔹 2. Structure of a Recursive Function

Every recursive function has **two main parts**:

| Part | Description |
|---|---|
| **Base Case** | The condition where the function stops calling itself. |
| **Recursive Case** | The part where the function calls itself with a smaller input. |

### 🧠 General Syntax

```
def recursive_function(parameters):
    if base_condition:
        return some_value       # Base Case
    else:
        return recursive_function(modified_parameters)  # Recursive Case
```

## 🔹 3. Example: Countdown Using Recursion

Example:

```
def countdown(n):
    if n == 0:  # Base Case
        print("Time's up!")
    else:
        print(n)
        countdown(n - 1)  # Recursive Case


countdown(5)
```

### ✅ Output:

```
5
4
3
2
1
Time's up!
```

**🧩 Explanation:**

- countdown(5) calls countdown(4)
- countdown(4) calls countdown(3)
- ... until n == 0, where recursion stops.

## 🔹 4. Why Use Recursion?

Recursion is useful for:

- Solving problems that can be divided into smaller similar sub-problems
- Traversing structures like trees, directories, and linked lists
- Solving mathematical problems like factorial, Fibonacci, etc.

However:
👉 Recursion can be **slower** and **use more memory** than loops if not used carefully.

## 🔷 5. Example: Factorial Using Recursion

**Factorial (n!) = n × (n−1) × (n−2) × ... × 1**

### Example:

```
def factorial(n):
    if n == 1:   # Base Case
        return 1
    else:
        return n * factorial(n - 1)  # Recursive Case


print("Factorial of 5 is:", factorial(5))
```

### ✅ Output:

```
Factorial of 5 is: 120
```

### 🧠 Step-by-Step Execution (Recursive Tree):

```
factorial(5)
= 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 * (3 * factorial(2)))
= 5 * (4 * (3 * (2 * factorial(1))))
= 5 * (4 * (3 * (2 * 1)))
= 120
```

🧩 **Base Case:** factorial(1) returns 1
🧩 **Recursive Case:** Each call waits for the next call to finish and multiplies the result.

## 🔷 6. Example: Sum of Natural Numbers

**Formula:** 1 + 2 + 3 + ... + n

Example:

```
def add_numbers(n):
  if n == 0:
    return 0
  else:
    return n + add_numbers(n - 1)


print("Sum of first 5 numbers is:", add_numbers(5))
```

### ✅ Output:

```
Sum of first 5 numbers is: 15
```

### 🧠 How It Works:

add_numbers(5)

= 5 + add_numbers(4)

= 5 + (4 + add_numbers(3))

= 5 + (4 + (3 + add_numbers(2)))

= 5 + (4 + (3 + (2 + add_numbers(1))))

= 5 + 4 + 3 + 2 + 1 + 0

= 15

## 🔷 7. Example: Fibonacci Series Using Recursion

**Fibonacci Series:** 0, 1, 1, 2, 3, 5, 8, 13, ...

**Formula:**

f(0) = 0

f(1) = 1

f(n) = f(n-1) + f(n-2)

Example:

```
def fibonacci(n):
    if n <= 1:  # Base Case
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)  # Recursive Case

for i in range(8):
    print(fibonacci(i), end=" ")
```

### ✅ Output:

0 1 1 2 3 5 8 13

### ⚙️ How It Works:

- fibonacci(5) calls fibonacci(4) and fibonacci(3)
- Each of those calls further splits into smaller problems
- Continues until n <= 1

This builds a **recursion tree** that combines results step by step.

## 🔹 8. Advantages of Recursion

✅ Makes code **clean**, **simple**, and **closer to mathematical logic**
✅ Reduces code length compared to loops
✅ Best for problems like **tree traversal**, **search**, and **divide & conquer** algorithms (e.g., QuickSort, MergeSort)

## 🔹 9. Disadvantages of Recursion

⚠️ High **memory usage** — each function call stores data on the call stack
⚠️ **Slower** than loops for large data
⚠️ Risk of **RecursionError** if base condition is missing or incorrect

Example of Recursion Error:

```
def infinite_recursion():
    print("Hello")
    infinite_recursion()
```

infinite_recursion()

### ❌ Output:

RecursionError: maximum recursion depth exceeded

💡 Python has a limit on how many times a function can call itself (default is **1000 calls**).

You can check it using:

import sys

print(sys.getrecursionlimit())

And even set it (carefully) using:

sys.setrecursionlimit(2000)

## 🔹 10. Recursion vs Iteration

| Feature | Recursion | Iteration (Loop) |
|---|---|---|
| **Definition** | Function calling itself | Looping (for/while) |
| **Stopping Condition** | Base case | Loop condition |
| **Memory Use** | High (stack used) | Low |
| **Speed** | Slower | Faster |
| **Code Size** | Shorter, elegant | Longer |
| **Examples** | Factorial, Fibonacci, Tree traversal | Counting, summing |

## 🔹 11. Real-Life Example — Directory Traversal

Recursive logic is used in file systems to scan through folders and subfolders.

Example:

import os

```
def explore(path):
    for item in os.listdir(path):
        full_path = os.path.join(path, item)
        if os.path.isdir(full_path):
            explore(full_path)  # Recursive call
        else:
            print(full_path)

# explore("C:/Users/Naveen/Documents")
```

💡 The function calls itself for **each subdirectory** — classic recursion in real life.

## 🔹 12. Practice Questions

Try these to master recursion 👇

Write a recursive function to find the **factorial** of a number.
Write a recursive function to find the **sum of digits** of a number.
Write a recursive function to print the **Fibonacci sequence** up to n terms.
Write a recursive function to reverse a string.
Write a recursive function to calculate the **power** of a number ($x^n$).

## 🔹 13. Real-Life Analogy

Imagine **standing in front of mirrors** — each mirror reflects another mirror infinitely.
Each reflection represents a **function calling itself**, and the smallest visible reflection represents the **base case** where recursion stops.

## 🔹 14. Summary Table

| Term | Description | Example |
|---|---|---|
| **Recursion** | Function calling itself | f() calls f() |
| **Base Case** | Stops recursion | if n == 0: |
| **Recursive Case** | Calls itself with smaller input | return n * f(n-1) |
| **RecursionError** | Too many recursive calls | No base case |
| **Uses** | Factorial, Fibonacci, Tree traversal | Divide & Conquer |

### 🔚 Conclusion

Recursion is one of Python's **most elegant and powerful** programming techniques.
It allows you to solve problems by **dividing them into smaller parts**, but it must always include a **base case** to avoid infinite loops.

🧠 "Recursion is thinking smaller — solving a small part of the problem and letting the function handle the rest."

## Day 38: Exception Handling - Try, Except

## ✅ Day 38: Exception Handling in Python – Try and Except

### 🎯 Objective:

Understand what **exceptions** are in Python, why they occur, and how to handle them using **try** and **except** blocks.

By the end of this lesson, you'll learn how to make your programs **error-proof** and **crash-free**, just like professional developers do. 🚀

### 🔹 1. What is an Exception?

An **exception** is an **error that occurs during program execution** — which interrupts the normal flow of your program.

When Python encounters an unexpected situation (like dividing by zero, using an undefined variable, etc.), it **raises an exception**.

If you don't handle it, the program **crashes** and stops running. ❌

🧩 **Example Without Handling:**

```
num1 = 10

num2 = 0

result = num1 / num2

print("Result:", result)
```

❌ **Output:**

```
ZeroDivisionError: division by zero
```

💡 **Explanation:**
You can't divide by zero, so Python raises a **ZeroDivisionError**.

To prevent the program from crashing, we can **handle** this using try and except.

## 🔹 2. What is Exception Handling?

**Exception handling** means **anticipating errors** and writing code to deal with them, so the program can continue running smoothly.

Python provides several keywords for this:

| Keyword | Purpose |
| --- | --- |
| **try** | Write the code that might cause an exception |
| **except** | Handle the exception (error) |
| **else** | Runs if no exception occurs |
| **finally** | Always runs (used for cleanup, closing files, etc.) |

👉 In this lesson, we'll focus on **try** and **except**.

### 🔹 3. Basic Structure of Try–Except

✅ **Syntax:**

```
try:
    # code that may cause an exception
except:
    # code to handle the exception
```

🧩 **Example 1:**

```
try:
    num1 = 10
    num2 = 0
    print(num1 / num2)
except:
```

```
    print("Oops! Division by zero is not allowed.")
```

✅ **Output:**

Oops! Division by zero is not allowed.

💡 The program didn't crash — instead, the except block caught the error and printed a friendly message.

## 🔹 4. Handling Specific Exceptions

It's a good practice to **catch specific exceptions**, not all errors blindly.
That way, you can handle each type of error differently.

🧩 **Example 2:**

```
try:
    number = int(input("Enter a number: "))
    print("Result:", 10 / number)
except ZeroDivisionError:
    print("You cannot divide by zero.")
except ValueError:
    print("Please enter a valid number.")
```

✅ **Outputs:**

If user enters 0 👇

You cannot divide by zero.

If user enters abc 👇

Please enter a valid number.

💡 Each exception type (ZeroDivisionError, ValueError) is handled **separately**.

### 🔹 5. Catching Multiple Exceptions Together

You can also handle **multiple exceptions in one line** using parentheses ().

🧩 **Example 3:**

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except (ZeroDivisionError, ValueError):
    print("Error: Invalid input or division by zero.")
```

✅ This is useful when different exceptions require **the same handling**.

### 🔹 6. Using Exception Object

You can also **capture the error message** using the **as** keyword.

🧩 **Example 4:**

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    result = a / b
    print("Result:", result)
except Exception as e:
    print("An error occurred:", e)
```

✅ **Output:**

An error occurred: division by zero

💡 Here, e contains the **actual error message** generated by Python.

## 🔷 7. Try–Except Flow Diagram

### 🧭 Flow of Execution:

Python runs the code inside try.
If **no error**, the except block is skipped.
If **error occurs**, Python jumps to except.
Program continues normally after that.

🧠 **Example 5:**

```
try:
    print("Start of try block")
    x = 1 / 0
    print("This line will not run")
except:
    print("Exception handled")
print("Program continues...")
```

✅ **Output:**

Start of try block

Exception handled

Program continues...

💡 Once an error occurs, Python **immediately jumps to except** and skips the rest of the try block.

## 🔹 8. Nested Try–Except Blocks

You can also **nest** try–except blocks if you want to handle different errors at different code levels.

🧩 **Example 6:**

```
try:
    a = int(input("Enter number: "))
    try:
        b = int(input("Enter another number: "))
        print("Result:", a / b)
    except ZeroDivisionError:
        print("Division by zero not allowed.")
except ValueError:
    print("Please enter only numbers.")
```

✅ This allows more **fine-grained error control**.

## 🔹 9. Example – Real-Life Scenario (File Handling)

Let's say you're reading from a file that may not exist.

🧩 **Example 7:**

```
try:
    file = open("data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
```

```
    print("The file you are trying to read does not exist.")
```

✅ **Output:**

The file you are trying to read does not exist.

💡 The program won't crash even if the file is missing.

### 🔹 10. Multiple Except Blocks for Different Errors

🧩 **Example 8:**

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    print("Result:", num1 / num2)
except ZeroDivisionError:
    print("You cannot divide by zero.")
except ValueError:
    print("Please enter numbers only.")
except Exception as e:
    print("Unexpected error:", e)
```

✅ The **last except** (Exception as e) catches **any unhandled errors**.

### 🔹 11. Why Exception Handling is Important

| Without Exception Handling | With Exception Handling |
| --- | --- |
| Program crashes on error | Program continues safely |
| No user-friendly messages | Custom, meaningful messages |
| Debugging is harder | Easier to find and fix issues |
| Poor user experience | Professional-grade reliability |

## 🔹 12. Common Exceptions in Python

| Exception Type | Description |
| --- | --- |
| ZeroDivisionError | Dividing by zero |
| ValueError | Invalid value (e.g., converting “abc” to int) |
| TypeError | Operation on wrong data type |
| FileNotFoundError | File not found |
| IndexError | Accessing index that doesn’t exist |
| KeyError | Accessing missing key in dictionary |
| NameError | Using a variable that hasn’t been defined |
| ImportError | Importing a module that doesn’t exist |

## 🔹 13. Example – Calculator with Exception Handling

🧩 **Example 9:**

```
try:
    a = float(input("Enter first number: "))
    b = float(input("Enter second number: "))
    operation = input("Enter operation (+, -, *, /): ")
```

```
    if operation == "+":
        print("Result:", a + b)
    elif operation == "-":
        print("Result:", a - b)
    elif operation == "*":
        print("Result:", a * b)
    elif operation == "/":
        print("Result:", a / b)
    else:
        print("Invalid operation.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Please enter valid numbers only.")
except Exception as e:
    print("Something went wrong:", e)
```

✅ This program can handle **multiple types of user errors gracefully**.

## 🔹 14. Practice Questions

🧠 Try these by yourself 👇

Write a program to take two numbers as input and handle ZeroDivisionError.

Handle ValueError when converting input to integer.

Create a program to open a file and handle FileNotFoundError.

Write a program to access a list index and handle IndexError.

Combine multiple exceptions in one line using (ZeroDivisionError, ValueError).

## 🔹 15. Summary Table

| Keyword | Description | Example |
|---|---|---|
| **try** | Code that may cause error | try: x = 1/0 |
| **except** | Handles the error | except: print("Error") |
| **except Exception as e** | Captures error message | except Exception as e: |
| **Multiple excepts** | Handle multiple errors separately | except ValueError: ... except ZeroDivisionError: ... |

## 🔚 Conclusion

Exception handling is one of Python's **most powerful safety features**.

It ensures your program **doesn't crash** when something unexpected happens.

Always remember:

🧠 “It’s better to **handle** an error gracefully than to let your program crash silently.”

Day 39: Else, Finally Blocks

## ✅ Day 39: Exception Handling in Python – else and finally Blocks

**🎯 Objective:**

By the end of this lesson, you’ll clearly understand:

- What are **else** and **finally** blocks in Python exception handling
- When and why to use them
- How they make your programs more professional and safe

We’ll cover step-by-step explanations, real-life examples, syntax, and small projects. 🚀

### 🔹 1. Quick Recap: What We Learned So Far

In the previous lesson (Day 38), we learned about:

- **try block:** Contains code that may cause an exception
- **except block:** Catches and handles errors gracefully

Now, let’s move one step further with two new blocks:
👉 **else** and **finally**

### 🔹 2. else Block – What and Why?

The **else block** runs **only if no exception occurs** in the try block.

Think of it as a “**success message**” — it executes **when everything goes right**.

✅ **Syntax:**

```
try:
    # code that may cause an error
except:
    # handles error
else:
    # runs if no error occurs
```

🧩 **Example 1:**

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful! Result =", result)
```

✅ **Output 1 (if user enters 10 and 2):**

```
Division successful! Result = 5.0
```

✅ **Output 2 (if user enters 10 and 0):**

```
Cannot divide by zero!
```

💡 **Explanation:**

- If the division runs fine → else executes.
- If an exception occurs → else block is **skipped**.

🧠 **When to Use else:**

Use the else block when you want to:

- Run code **only if no errors occur**
- Keep success logic **separate** from error-handling logic

## 🔹 3. finally Block – What and Why?

The **finally block** executes **no matter what happens** —
✅ whether there is an exception or not.

It's typically used for **clean-up actions** like:

- Closing files or databases
- Releasing memory or network connections
- Printing final messages

✅ **Syntax:**

```
try:
    # code that may cause an error
except:
    # handle error
finally:
    # code that always executes
```

🧩 **Example 2:**

```
try:
    print("Opening file...")
    file = open("example.txt", "r")
```

```
    data = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("Closing file (if open).")
```

✅ **Output:**

```
Opening file...
File not found!
Closing file (if open).
```

💡 Even though the file doesn't exist, finally block still runs.

🔷 **4. try + except + else + finally Together

You can combine **all four blocks** for maximum control and reliability.

🧩 **Example 3:**

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Division by zero!")
except ValueError:
    print("Error: Invalid input!")
```

```
else:
    print("Result =", result)
finally:
    print("Execution complete. Thank you!")
```

✅ **Outputs:**

**Case 1 – Valid Input:**

Enter first number: 10

Enter second number: 2

Result = 5.0

Execution complete. Thank you!

**Case 2 – Division by Zero:**

Enter first number: 10

Enter second number: 0

Error: Division by zero!

Execution complete. Thank you!

**Case 3 – Invalid Input:**

Enter first number: abc

Error: Invalid input!

Execution complete. Thank you!

💡 The finally block **always runs**, even when exceptions occur.

🔹 **5. Flow of Execution – Step by Step**

Here's how Python executes the blocks 👇

| Step | What Happens | Example |
|---|---|---|
| | Executes the try block | Possible error may occur |
| | If an exception occurs → jumps to except | Handles the error |
| | If no exception → executes else | Shows success |
| | Always executes finally | Clean-up / wrap-up |

🧩 **Flow Example 4:**

```
try:
    print("Step 1: Trying...")
    x = 10 / 2
except:
    print("Step 2: Exception occurred!")
else:
    print("Step 3: No errors detected!")
finally:
    print("Step 4: Always executed!")
```

✅ **Output:**

Step 1: Trying...

Step 3: No errors detected!

Step 4: Always executed!

If you replace 10 / 2 with 10 / 0, then output will be 👇

Step 1: Trying...

Step 2: Exception occurred!

Step 4: Always executed!

💡 The finally block runs **in both cases**.

## 🔹 6. Real-Life Example – File Handling

Let’s use try, except, and finally in a realistic example.

🧩 **Example 5:**

```
try:
    file = open("data.txt", "r")
    content = file.read()
    print("File Content:\n", content)
except FileNotFoundError:
    print("Error: The file does not exist.")
else:
    print("File read successfully!")
finally:
    print("Closing file...")
    try:
```

```
        file.close()
    except:
        pass
```

✅ **Output 1 (if file exists):**

File Content:

Hello, Python!

File read successfully!

Closing file...

✅ **Output 2 (if file not found):**

Error: The file does not exist.

Closing file...

💡 The finally block ensures that the file is **closed** even if an error occurs.

### 🔹 7. Why Use finally?

| Scenario | Why finally helps |
| --- | --- |
| Working with files | Ensures the file is closed |
| Database connections | Ensures connection is closed properly |
| Network requests | Ensures disconnection after use |
| Critical cleanup | Releases system resources safely |

## 🔹 8. Example – Banking Transaction Simulation

🧩 **Example 6:**

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

    if withdraw > balance:
        raise ValueError("Insufficient balance!")

    balance -= withdraw
except ValueError as e:
    print("Transaction failed:", e)
else:
    print("Transaction successful! Remaining balance:", balance)
finally:
    print("Session closed. Please remove your card.")
```

✅ **Output 1:**

```
Enter amount to withdraw: 3000
Transaction successful! Remaining balance: 2000
Session closed. Please remove your card.
```

✅ **Output 2:**

Enter amount to withdraw: 6000

Transaction failed: Insufficient balance!

Session closed. Please remove your card.

💡 finally ensures that **"Session closed"** message appears no matter what happens.

### 🔹 9. Example – Program Always Ends Gracefully

🧩 **Example 7:**

```
try:
    print("Starting program...")
    x = int(input("Enter a number: "))
    print("Square:", x * x)
except ValueError:
    print("Invalid input. Please enter a number.")
finally:
    print("Program finished. Exiting safely...")
```

✅ The program always ends gracefully — even if input is wrong.

### 🔹 10. Nested try-finally Example

You can also **nest** finally blocks for different parts of code.

🧩 **Example 8:**

```
try:
    print("Outer try block")
    try:
```

```
        print("Inner try block")
        x = 1 / 0
    finally:
        print("Inner finally always runs!")
except ZeroDivisionError:
    print("Handled ZeroDivisionError in outer block.")
finally:
    print("Outer finally always runs!")
```

✅ **Output:**

```
Outer try block
Inner try block
Inner finally always runs!
Handled ZeroDivisionError in outer block.
Outer finally always runs!
```

💡 Every finally executes in its own scope — no matter what happens.

## 🔹 11. Difference Between else and finally

| Feature | else | finally |
|---|---|---|
| When it runs | Only if **no exception** occurs | Always runs (error or not) |
| Typical use | For success message / logic | For cleanup (close file, release resource) |
| Skipped if error? | Yes | No |
| Example use | Print result if no error | Close file, release memory |

## 🔹 12. Summary Table

| Block | Purpose | Executes When |
|---|---|---|
| **try** | Code that may cause error | Always first |
| **except** | Handles the error | When error occurs |
| **else** | Runs only if no error | After try succeeds |
| **finally** | Runs always | Whether error or not |

## 🔹 13. Practice Tasks

💪 Try these on your own 👇

Write a program that divides two numbers using all four blocks (try, except, else, finally).

Open a file, read its content, handle errors, and close it properly using finally.

Simulate an ATM system where finally prints “Card Ejected.”

Create a nested try-finally example and trace which one executes first.

Experiment by raising a manual exception inside try and see which blocks execute.

## 🔹 14. Real-Life Analogy

🧠 Think of this like a **restaurant process**:

| Stage | Equivalent Python Block |
|---|---|
| Cooking the food | try |
| Handling mistakes (burnt dish) | except |
| Serving if perfect | else |
| Cleaning kitchen (end of day) | finally |

Even if cooking fails or goes great, the kitchen **always needs cleaning!** 🍽️

### 🔚 Conclusion

- else helps you **run code when everything goes right**
- finally ensures **cleanup happens no matter what**

- Together with try and except, they make your program **robust, reliable, and professional**

Day 40: Built-in Functions Overview

# 🧠 Day 40: Built-in Functions in Python — Complete Overview

## 🎯 Objective:

By the end of this lesson, you'll clearly understand:

✅ What built-in functions are
✅ Why Python provides them
✅ How to use the most important built-in functions
✅ Real-time examples for each category

## 🔹 1. Introduction – What Are Built-in Functions?

Python is known for its **simplicity and power**, and one of the main reasons for that is — it comes with **hundreds of built-in functions**.

These are **ready-made functions** that you can use **without importing any library**.

👉 **You don't have to define them — they are always available!**

### 🧩 Definition:

A **built-in function** is a function that is pre-defined in Python and can be used directly in your code without importing any module.

### 💬 Example:

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

x = len("Python")

y = max(5, 10, 15)

print(x, y)
```

✅ **Output:**

Hello, Naveen!

6 15

Here,

- print() → displays output
- len() → finds length of a string
- max() → returns the maximum value

💡 You didn't define any of these — Python gave them to you for free!

## 🔹 2. Why Are Built-in Functions Important?

✔️ Save time — no need to write extra code
✔️ Improve performance — optimized internally by Python
✔️ Increase readability — easily understood by everyone
✔️ Always available — no import required

## 🔹 3. Categories of Built-in Functions

Python's built-in functions can be grouped as follows 👇

| Category | Examples |
| --- | --- |
| Input/Output | print(), input() |
| Type Conversion | int(), float(), str(), list(), tuple() |
| Mathematical | abs(), pow(), round(), min(), max(), sum() |
| Sequence Related | len(), sorted(), reversed(), enumerate(), zip() |
| Logical | any(), all() |
| Utility / Information | id(), type(), dir(), help() |
| Object Handling | isinstance(), issubclass(), callable() |
| Miscellaneous | eval(), exec(), hash(), format() |

Let’s explore each category step by step 🔽

## 🧮 4. Input and Output Functions

### 🧩 Example 1:

name = input("Enter your name: ")

print("Hello,", name)

✅ **Output:**

Enter your name: Naveen

Hello, Naveen

- input() → Takes user input as a string.

- print() → Displays output to the screen.

## 🔢 5. Type Conversion Functions

These functions convert data from one type to another.

| Function | Description | Example | Output |
|---|---|---|---|
| int() | Converts to integer | int("10") | 10 |
| float() | Converts to float | float(5) | 5.0 |
| str() | Converts to string | str(100) | '100' |
| list() | Converts to list | list("abc") | ['a', 'b', 'c'] |
| tuple() | Converts to tuple | tuple([1,2,3]) | (1, 2, 3) |
| set() | Converts to set | set([1,1,2]) | {1,2} |
| dict() | Converts to dictionary | dict(a=1,b=2) | {'a':1,'b':2} |

## ➕ 6. Mathematical Functions

| Function | Description | Example | Output |
|---|---|---|---|
| abs() | Absolute value | abs(-10) | 10 |
| pow(x, y) | Power (x^y) | pow(2,3) | 8 |
| round() | Rounds to nearest integer | round(3.6) | 4 |
| min() | Returns smallest value | min(5,10,1) | 1 |
| max() | Returns largest value | max(5,10,1) | 10 |
| sum() | Adds all elements in iterable | sum([1,2,3]) | 6 |

🧩 **Example 2:**

```
numbers = [4, 7, 2, 9]

print("Max:", max(numbers))

print("Min:", min(numbers))

print("Sum:", sum(numbers))

print("Average:", sum(numbers) / len(numbers))
```

✅ **Output:**

```
Max: 9
```

Min: 2

Sum: 22

Average: 5.5

## 📚 7. Sequence and Iterable Functions

### 🧩 len() – Length of an Object

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

print(len(fruits))

✅ Output:

3

### 🧩 sorted() – Sort Items

numbers = [3, 1, 5, 2]

print(sorted(numbers))

✅ Output:

[1, 2, 3, 5]

### 🧩 reversed() – Reverse Order

for i in reversed(range(5)):

  print(i, end=" ")

✅ Output:

4 3 2 1 0

### 🧩 enumerate() – Add Counter

for index, value in enumerate(["Python", "SQL", "Power BI"]):

  print(index, value)

✅ Output:

0 Python

1 SQL

2 Power BI

### 🧩 zip() – Combine Iterables

names = ["Naveen", "Ravi", "John"]

scores = [85, 90, 88]

for n, s in zip(names, scores):

  print(n, "->", s)

✅ Output:

Naveen -> 85

Ravi -> 90

John -> 88

## ⚙️ 8. Logical Functions

| Function | Description | Example | Output |
|---|---|---|---|
| any() | Returns True if any element is True | any([0, 1, 0]) | True |
| all() | Returns True if all elements are True | all([1, 1, 1]) | True |
| all([1, 0, 1]) | → False | | |

## 🧾 9. Utility and Information Functions

| Function | Description | Example | Output |
|---|---|---|---|
| id() | Returns unique ID of an object | id(10) | Some unique number |
| type() | Returns object's type | type("Naveen") | <class 'str'> |
| dir() | Returns list of attributes/methods | dir(str) | Shows all methods of string |
| help() | Provides documentation help | help(print) | Displays info about print() |

🧩 **Example:**

```
x = "Hello"

print(type(x))

print(id(x))

print(dir(x)[:5])  # Show first 5 methods
```

✅ Output (example):

```
<class 'str'>

140703282654224

['__add__', '__class__', '__contains__', '__delattr__', '__dir__']
```

## 👨‍🏫 10. Object and Class Functions

| Function | Description | Example | Output |
|---|---|---|---|
| isinstance(x, type) | Checks if object is instance of a class | isinstance(5, int) | True |
| issubclass(A, B) | Checks if class A inherits from class B | issubclass(bool, int) | True |
| callable() | Checks if object is callable | callable(print) | True |

🧩 **Example:**

```
print(isinstance(10, int))     # True
print(issubclass(bool, int))   # True
print(callable(print))         # True
```

## 🧮 11. Evaluation and Execution Functions

| Function | Description | Example | Output |
|---|---|---|---|
| eval() | Evaluates a string expression | eval("5 + 10") | 15 |
| exec() | Executes Python code dynamically | exec("x=5; print(x**2)") | 25 |

⚠️ Note:

Be careful with eval() and exec() — they can run arbitrary code, so don't use them with untrusted input.

## 💰 12. Formatting and Hash Functions

| Function | Description | Example | Output |
|---|---|---|---|
| format() | Returns formatted string | "{} scored {}".format("Naveen", 90) | 'Naveen scored 90' |
| hash() | Returns hash value of an object | hash("Python") | (unique integer) |

🧩 **Example:**

```
name = "Naveen"

score = 95

print("{} got {} marks.".format(name, score))
```

print("Hash of name:", hash(name))

✅ Output:

Naveen got 95 marks.

Hash of name: 890765409324

## 📖 13. Most Commonly Used Built-in Functions

Here's a quick summary of the **most used functions** every Python programmer should know 👇

| Category | Function | Purpose |
|---|---|---|
| I/O | print(), input() | Output & Input |
| Type Conversion | int(), float(), str(), list() | Change data types |
| Math | abs(), pow(), round(), sum() | Basic math |
| Sequence | len(), sorted(), enumerate(), zip() | Work with lists/tuples |
| Logic | any(), all() | Boolean checks |
| Info | type(), id(), dir(), help() | Object info |
| Utility | eval(), exec(), hash(), format() | Code execution & formatting |

## 🧠 14. Real-Life Analogy

Think of Python built-in functions like **tools in a Swiss Army knife** 🛠️

- You don’t need to build a knife or screwdriver yourself — they’re already there.
- You just pick the right one for the job.

In the same way, built-in functions are your **ready-to-use tools** for any programming task.

## 💪 15. Practice Exercises

Try these small programs to strengthen your understanding 👇

Find the maximum, minimum, and sum of a list of numbers.
Convert user input from string to integer and double it.
Use zip() to combine names and marks into pairs.
Print the data type and ID of a variable.
Sort a list of names alphabetically using sorted().
Check if all numbers in a list are positive using all().
Evaluate a mathematical expression entered as a string using eval().

## 🧾 16. Summary Table

| Concept | Description | Example |
|---|---|---|
| Built-in Functions | Ready-made functions in Python | print(), len(), max() |
| Purpose | Save time, improve performance | sum([1,2,3]) → 6 |
| Always Available | No need to import | print("Hello") |
| Number of Built-ins | 70+ functions | Use dir(__builtins__) to list them |

### 🧩 Example: Get All Built-in Functions

```
import builtins

print(dir(builtins))
```

✅ Output: (shows all available functions like abs, len, max, sum, etc.)

## 🏁 17. Key Takeaways

✔ Built-in functions are **predefined and always available**
✔ They make coding **faster and easier**
✔ You can group them by type (I/O, math, sequence, info, etc.)
✔ Use dir(__builtins__) to explore them all
✔ Learn to **use them smartly** — they're your foundation for writing efficient code

### 🌟 In short:

Python's built-in functions are like your **first programming superpowers** — always ready, always reliable, and always there to help you write less and do more. ⚡

# Day 41: List Comprehension

## 🧠 Day 41: List Comprehension in Python

### 🎯 Concept Overview

**List Comprehension** is one of the most **powerful and concise** ways to create lists in Python. It allows you to build a new list by applying an **expression** to each item in an **iterable** (like a list, tuple, or string) — all in **a single line** of code.

In simple words:
👉 It's a **shorter and faster** way to create lists compared to using loops.

### 🧩 Syntax

new_list = [expression for item in iterable if condition]

**Explanation:**

- expression: The operation or transformation you want to perform.
- item: Each element taken from the iterable (like a list or range).
- iterable: The data source (like a list, tuple, or range).
- condition (optional): A filter to include only certain elements.

### 🔍 Example 1: Basic List Comprehension

**Traditional Way:**

```
numbers = [1, 2, 3, 4, 5]

squares = []

for num in numbers:

    squares.append(num ** 2)

print(squares)
```

**Output:**

```
[1, 4, 9, 16, 25]
```

**Using List Comprehension:**

```
numbers = [1, 2, 3, 4, 5]

squares = [num ** 2 for num in numbers]

print(squares)
```

**Output:**

```
[1, 4, 9, 16, 25]
```

👉 Both methods give the same result, but list comprehension is shorter and cleaner.

## 🎯 Example 2: Using a Condition

You can add an **if condition** inside list comprehension to filter elements.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

even_numbers = [num for num in numbers if num % 2 == 0]

print(even_numbers)
```

**Output:**

```
[2, 4, 6, 8]
```

## ⚙️ Example 3: Applying Operations

You can perform operations on elements before adding them to the new list.

```
numbers = [1, 2, 3, 4, 5]

doubled = [num * 2 for num in numbers]

print(doubled)
```

**Output:**

[2, 4, 6, 8, 10]

### 🧠 Example 4: Nested List Comprehension

You can even use **nested loops** inside a list comprehension.

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

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

print(flattened)

**Output:**

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

👉 It extracts all numbers from nested lists into a single flat list.

### 🔤 Example 5: String Operations

You can use it to modify strings as well.

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

uppercase_fruits = [fruit.upper() for fruit in fruits]

print(uppercase_fruits)

**Output:**

['APPLE', 'BANANA', 'CHERRY']

### 🧮 Example 6: With Range()

squares = [x ** 2 for x in range(1, 11)]

print(squares)

**Output:**

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

### ⚖️ Comparison: Traditional vs Comprehension

| Feature | Traditional Loop | List Comprehension |
|---|---|---|
| Code Length | Longer | Shorter |
| Speed | Slower | Faster |
| Readability | Clearer for beginners | Concise for experts |
| Flexibility | High | Moderate |

### 💡 Example 7: Condition with Else

You can also add an **if-else** condition.

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

even_odd = ["Even" if x % 2 == 0 else "Odd" for x in numbers]

print(even_odd)

**Output:**

['Odd', 'Even', 'Odd', 'Even', 'Odd']

### 🧩 Example 8: List of Tuples

You can easily create tuples inside a list.

pairs = [(x, y) for x in [1, 2, 3] for y in [4, 5, 6]]

print(pairs)

**Output:**

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

### ⚡ Real-Life Example: Filtering Emails

emails = ["john@gmail.com", "naveen@yahoo.com", "mike@gmail.com", "emma@outlook.com"]

gmail_users = [email for email in emails if "gmail.com" in email]

print(gmail_users)

**Output:**

['john@gmail.com', 'mike@gmail.com']

### 🧠 Key Points to Remember

✅ List comprehensions are **faster** and **more readable** for short tasks.
✅ You can include **conditions** and **nested loops**.
✅ Keep them simple — avoid overcomplicating with too many conditions.
✅ Always use them when creating a list from another iterable.

### 🏁 Summary

| Concept | Description |
|---|---|
| **Definition** | A concise way to create lists using a single line of code |
| **Syntax** | [expression for item in iterable if condition] |
| **Best Use** | Creating new lists by transforming or filtering existing iterables |
| **Key Benefit** | Shorter, cleaner, and often faster code |

### 🧩 Mini Practice Tasks

Create a list of squares for numbers 1–10 using list comprehension.
Generate a list of even numbers from 1–20.
Create a list of words that start with the letter ‘A’ from a given list.
Flatten a 2D list using list comprehension.
Convert all strings in a list to lowercase.

Day 42: Dictionary and Set Comprehension

# 🧠 Day 42: Dictionary and Set Comprehension in Python

Comprehensions make your Python code **faster, cleaner, and more readable**.
You've already learned **List Comprehension** — now it's time to master **Dictionary** and **Set Comprehension**, which follow the same idea.

# 🔹 1. Dictionary Comprehension

## 🎯 Concept Overview

**Dictionary Comprehension** provides a **concise way** to create dictionaries from iterables in a single line of code.

It's used to **transform**, **filter**, or **map** data into key-value pairs easily.

## 🧩 Syntax

new_dict = {key_expression: value_expression for item in iterable if condition}

**Explanation:**

- key_expression: Defines what each key will be.
- value_expression: Defines what each value will be.
- iterable: A list, range, or any collection to iterate over.
- condition (optional): Filters which items to include.

## 🧠 Example 1: Basic Dictionary Comprehension

**Goal:** Create a dictionary of numbers and their squares.

**Traditional Way:**

```
squares = {}

for x in range(1, 6):

   squares[x] = x * x

print(squares)
```

**Output:**

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

**Using Dictionary Comprehension:**

```
squares = {x: x ** 2 for x in range(1, 6)}

print(squares)
```

**Output:**

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

✅ Shorter, cleaner, and easier to read!

## ⚙️ Example 2: Conditional Dictionary

You can include a condition to filter items.

```
even_squares = {x: x ** 2 for x in range(1, 11) if x % 2 == 0}

print(even_squares)
```

**Output:**

{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}

👉 Only even numbers are included.

## 💡 Example 3: Swapping Keys and Values

If you want to **invert a dictionary** (swap keys and values):

```
students = {'A': 90, 'B': 85, 'C': 92}

inverted = {value: key for key, value in students.items()}

print(inverted)
```

**Output:**

{90: 'A', 85: 'B', 92: 'C'}

## 🔤 Example 4: Working with Strings

```
word = "hello"

count = {char: word.count(char) for char in word}

print(count)
```

**Output:**

{'h': 1, 'e': 1, 'l': 2, 'o': 1}

👉 Quickly creates a character frequency dictionary!

## 🧮 Example 5: Using if-else in Comprehension

You can apply **if-else** logic directly inside dictionary comprehension.

```
numbers = range(1, 6)
```

```
result = {x: ('Even' if x % 2 == 0 else 'Odd') for x in numbers}

print(result)
```

**Output:**

```
{1: 'Odd', 2: 'Even', 3: 'Odd', 4: 'Even', 5: 'Odd'}
```

### ⚡ Example 6: Nested Comprehension

You can even create nested dictionaries!

```
matrix = {i: {j: i * j for j in range(1, 4)} for i in range(1, 4)}

print(matrix)
```

**Output:**

```
{1: {1: 1, 2: 2, 3: 3},

 2: {1: 2, 2: 4, 3: 6},

 3: {1: 3, 2: 6, 3: 9}}
```

### ⚖️ Advantages of Dictionary Comprehension

| Feature | Description |
|---|---|
| ✅ Conciseness | Creates dictionaries in one line |
| ⚡ Performance | Faster than normal loops |
| 💡 Readability | Easier to understand for simple transformations |
| 🧩 Flexibility | Supports if, else, and multiple loops |

## 🔹 2. Set Comprehension

### 🎯 Concept Overview

**Set Comprehension** works just like list comprehension,
but it creates a **set** instead of a list — meaning **no duplicate values**.

## 🧩 Syntax

new_set = {expression for item in iterable if condition}

**Explanation:**

- expression: What you want to store in the set.
- item: Each element from the iterable.
- condition (optional): A filter to include certain elements only.

## 🧠 Example 1: Basic Set Comprehension

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

unique_squares = {x ** 2 for x in numbers}

print(unique_squares)

**Output:**

{1, 4, 9, 16, 25}

👉 Notice duplicates are automatically removed.

## ⚙️ Example 2: With Condition

even_set = {x for x in range(1, 11) if x % 2 == 0}

print(even_set)

**Output:**

{2, 4, 6, 8, 10}

## 💡 Example 3: String Set Comprehension

word = "programming"

letters = {ch for ch in word if ch not in 'aeiou'}

print(letters)

**Output:**

{'r', 'g', 'm', 'p', 'n'}

👉 All unique consonants are extracted.

### ⚡ Example 4: Applying Operations

squares_set = {x * x for x in range(1, 6)}

print(squares_set)

**Output:**

{1, 4, 9, 16, 25}

### ⚖️ Advantages of Set Comprehension

| Feature | Description |
| --- | --- |
| ✅ Removes Duplicates | Automatically stores only unique values |
| ⚡ Fast | Faster than using loops |
| 💡 Simple | Easy to write and read |
| 🧠 Memory Efficient | Does not store duplicate elements |

## 💥 Real-Life Example:

✅ Dictionary Comprehension Example

Create a dictionary with names and lengths:

names = ["Naveen", "Raj", "Sita", "John"]

name_length = {name: len(name) for name in names}

print(name_length)

**Output:**

{'Naveen': 6, 'Raj': 3, 'Sita': 4, 'John': 4}

✅ Set Comprehension Example

Get unique email domains:

```
emails = ["a@gmail.com", "b@yahoo.com", "c@gmail.com", "d@outlook.com"]

domains = {email.split('@')[1] for email in emails}

print(domains)
```

**Output:**

```
{'gmail.com', 'yahoo.com', 'outlook.com'}
```

## 🧩 Key Differences

| Feature | Dictionary Comprehension | Set Comprehension |
|---|---|---|
| **Brackets** | {key: value ...} | {expression ...} |
| **Output Type** | Dictionary | Set |
| **Duplicates** | Keys must be unique | Values automatically unique |
| **Purpose** | Key-value pairs | Unique single values |

## 🏁 Summary

| Concept | Description |
| --- | --- |
| **Dictionary Comprehension** | Used to create key-value pairs in one line |
| **Set Comprehension** | Used to create unique value collections in one line |
| **Syntax** | {key: value for ...} (Dict), {value for ...} (Set) |
| **Advantages** | Shorter, cleaner, and more efficient than loops |
| **Best For** | Data transformation, filtering, and mapping |

### 🧠 Mini Practice Tasks

Create a dictionary of numbers and their cubes using comprehension.
Filter out names with less than 4 letters using dictionary comprehension.
Create a set of vowels found in the word "Education".
Make a set of even numbers from 1–20 using set comprehension.
Create a dictionary that maps each word to its uppercase version.

Day 43: Introduction to Modules and Import

# 🧠 Day 43: Introduction to Modules and Import in Python

### 🎯 Concept Overview

When you start writing Python programs, you'll often need to reuse code — like mathematical operations, random number generation, or date handling.
Instead of writing everything from scratch every time, Python gives you a powerful feature called **Modules**.

A **Module** in Python is simply a **file that contains Python code** — functions, variables, or classes — that you can use in other programs.

Think of a **module** as a **toolbox** that contains ready-made tools (functions and classes) you can use whenever you need.

## 🧩 What is a Module?

🟦 **Definition:**
A module is a Python file (.py) that contains definitions of **functions**, **variables**, and **classes** that you can import and use in another Python file.

👉 Example:
If you create a file named **math_operations.py**, it becomes a module that you can import into another Python program.

## ⚙️ Why Use Modules?

✅ To **reuse code** instead of rewriting it.
✅ To **organize code** into smaller, manageable files.
✅ To **maintain and debug** code easily.
✅ To **share and import** useful Python functions or packages.

## 📂 Types of Modules in Python

| Type | Description | Example |
|---|---|---|
| **Built-in Modules** | Already included in Python | math, random, os, datetime |
| **User-defined Modules** | Created by you | my_module.py |
| **External Modules** | Installed using pip | pandas, numpy, requests |

# 🔹 Built-in Modules

Python provides many **predefined modules** that come with Python installation.

Example: math, random, os, sys, datetime, json, etc.

## Example 1: Using the math Module

```
import math


print(math.sqrt(25))     # Square root
print(math.pi)           # Pi constant
print(math.factorial(5))  # Factorial
```

**Output:**

```
5.0
3.141592653589793
120
```

🧠 **Explanation:**

- math.sqrt(25) → returns the square root of 25
- math.pi → gives the value of π
- math.factorial(5) → returns 5 × 4 × 3 × 2 × 1 = 120

## Example 2: Using the random Module

```
import random


print(random.randint(1, 10))  # Random number between 1 and 10
print(random.choice(['apple', 'banana', 'cherry']))  # Random choice
```

**Output (varies):**

```
7
banana
```

## 🔹 Importing Modules

There are multiple ways to import and use modules in Python.

### a) Import the entire module

```
import math

print(math.sqrt(16))
```

✅ You must use the module name (math.) before accessing any function.

### b) Import specific functions

```
from math import sqrt, pi

print(sqrt(16))

print(pi)
```

✅ You can use the functions directly without writing math. each time.

### c) Import all functions (not recommended)

```
from math import *

print(sqrt(16))

print(pi)
```

⚠️ Not recommended for big projects — can cause name conflicts if multiple functions have the same name.

### d) Import with alias (short name)

```
import math as m

print(m.sqrt(25))
```

✅ Useful for large module names — makes code shorter and cleaner.

## 🔹 User-Defined Modules

You can create your own module and use it in other Python programs.

Step — Create a module file

**File name:** my_module.py

```
def greet(name):
```

```
    return f"Hello, {name}! Welcome to Python."


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

Step — Import and use it in another file

**File name:** main.py

```
import my_module


print(my_module.greet("Naveen"))
print(my_module.add(10, 20))
```

**Output:**

```
Hello, Naveen! Welcome to Python.
30
```

✅ You've successfully created and imported your own module!

## 🔹 Renaming (Aliasing) User Modules

You can rename your module while importing:

```
import my_module as m


print(m.add(5, 10))
```

**Output:**

```
15
```

## 🔹 Importing Specific Functions from User Modules

```
from my_module import greet


print(greet("Naveen"))
```

**Output:**

Hello, Naveen! Welcome to Python.

## 🔹 Using the dir() Function

The **dir()** function lists all names (functions, variables, etc.) defined inside a module.

Example:

```
import math

print(dir(math))
```

**Output (partial):**

['acos', 'asin', 'atan', 'ceil', 'cos', 'factorial', 'floor', 'sqrt', ...]

## 🔹 Using the help() Function

Use help() to get detailed documentation about a module or function.

```
import math

help(math.sqrt)
```

**Output:**

Help on built-in function sqrt in module math:

sqrt(x, /)

    Return the square root of x.

## 🧠 Importing from a Package (Folder)

A **package** is a folder containing multiple modules.
Each folder must contain a special file named **__init__.py** (can be empty).

Example:

```
my_package/
    __init__.py
    module1.py
    module2.py
```

Then you can import like this:

from my_package import module1

## ⚡ External Modules (Using pip)

If a module isn't built-in, you can install it from the internet using **pip (Python Package Installer)**.

Example:

pip install numpy

Then use it in Python:

import numpy as np

print(np.array([1, 2, 3]))

## 🧩 1 Commonly Used Built-in Modules

| Module | Purpose |
|---|---|
| math | Mathematical operations |
| random | Random number generation |
| datetime | Date and time operations |
| os | File and directory operations |
| sys | System-specific functions |
| json | Working with JSON data |
| statistics | Statistical calculations |
| platform | System information |

## 💡 Real-World Example

### Example 1: Using datetime Module

```
import datetime


current_time = datetime.datetime.now()

print("Current Date and Time:", current_time)
```

**Output:**

```
Current Date and Time: 2025-11-02 15:30:10.123456
```

### Example 2: Using os Module

```
import os


print(os.getcwd())  # Get current working directory

os.mkdir("test_folder")  # Create a new folder
```

**Output:**

```
C:\Users\Naveen\PythonProjects
```

## 🧠 Key Points to Remember

✅ A module is just a Python file (.py) with code that can be imported.
✅ Use import to bring in modules.
✅ Use as to give a shorter alias name.
✅ Use dir() and help() to explore modules.
✅ Built-in modules come with Python; external ones can be installed using pip.
✅ You can create your **own modules** to organize and reuse code.

## 🏁 Summary

| Concept | Description |
| --- | --- |
| **Module** | A Python file with reusable code |
| **Import** | Used to access module contents |
| **Built-in Modules** | Already available (e.g., math, os) |
| **User-defined Modules** | Created by you |
| **External Modules** | Installed using pip |
| **Key Functions** | dir(), help() |

## 🧩 Mini Practice Tasks

Import the math module and find the factorial of 6.

Create your own module named calc.py with add, subtract, multiply, and divide functions.

Use the datetime module to print today's date.

Install the requests module using pip and print its version.

Import only the sqrt and pi functions from math and calculate the area of a circle with radius 5.

# Day 44: Math and Random Modules

## ✅ Day 44: Math and Random Modules in Python

### 🎯 Learning Goal:

By the end of this lesson, you'll be able to:

- Use Python's **math** module for mathematical calculations.
- Use Python's **random** module to generate random numbers and choices.
- Understand the **difference between deterministic and random operations** in programming.

## 🧮 Introduction to the math Module

The math module in Python provides **mathematical functions and constants** like square root, power, trigonometric functions, etc.

To use it, we first import the module:

```
import math
```

### 🔹 Commonly Used math Module Functions

| Function | Description | Example | Output |
|---|---|---|---|
| math.sqrt(x) | Returns the square root of x | math.sqrt(16) | 4.0 |
| math.pow(x, y) | Returns x raised to the power y | math.pow(2, 3) | 8.0 |
| math.floor(x) | Rounds down to the nearest integer | math.floor(4.7) | 4 |
| math.ceil(x) | Rounds up to the nearest integer | math.ceil(4.2) | 5 |
| math.pi | Returns the constant π (pi) | math.pi | 3.1415926535... |
| math.e | Returns Euler’s number (e ≈ 2.718) | math.e | 2.718281828... |
| math.factorial(x) | Returns factorial of x | math.factorial(5) | 120 |
| math.fabs(x) | Returns the absolute value of x | math.fabs(-10) | 10.0 |

| | | | |
|---|---|---|---|
| math.log(x, base) | Returns log of x to given base | math.log(100, 10) | 2.0 |
| math.sin(x) | Returns sine of x (radians) | math.sin(math.pi/2) | 1.0 |
| math.cos(x) | Returns cosine of x | math.cos(0) | 1.0 |

🟦 **Example: Using Math Functions**

```
import math


num = 25

print("Square Root:", math.sqrt(num))

print("Power:", math.pow(2, 3))

print("Ceil:", math.ceil(4.1))

print("Floor:", math.floor(4.9))

print("Value of Pi:", math.pi)

print("Factorial:", math.factorial(5))
```

**Output:**

```
Square Root: 5.0

Power: 8.0

Ceil: 5

Floor: 4
```

Value of Pi: 3.141592653589793

Factorial: 120

## 🎲 Introduction to the random Module

The random module allows us to **generate random numbers** or **make random selections** — useful in games, simulations, and testing.

Import it using:

```
import random
```

🔹 **Commonly Used random Functions**

| Function | Description | Example | Output |
| --- | --- | --- | --- |
| random.random() | Returns a random float between 0 and 1 | random.random() | 0.5829... |
| random.randint(a, b) | Returns a random integer between a and b (inclusive) | random.randint(1, 10) | 7 |
| random.randrange(start, stop, step) | Returns a random number within a range | random.randrange(0, 10, 2) | 4 |
| random.choice(sequence) | Returns a random element from a list/tuple/string | random.choice(['red', 'blue', 'green']) | 'blue' |
| random.shuffle(list) | Randomly shuffles the list items | | |
| random.uniform(a, b) | Returns a random float | random.uniform(5, 10) | 7.65 |

| | between a | | |
|---|---|---|---|

🟦 **Example: Random Number Generation**

```
import random


print("Random Float:", random.random())

print("Random Integer (1–10):", random.randint(1, 10))

print("Random Number from Range (0–10 step 2):",
random.randrange(0, 10, 2))

print("Random Float between 5 and 10:", random.uniform(5, 10))
```

**Output:**

```
Random Float: 0.4638

Random Integer (1–10): 8

Random Number from Range (0–10 step 2): 6

Random Float between 5 and 10: 7.4512
```

🎮 **Example: Using random.choice() and random.shuffle()**

```
import random


colors = ['red', 'green', 'blue', 'yellow']


print("Random Color:", random.choice(colors))


random.shuffle(colors)
```

print("Shuffled Colors:", colors)

**Output:**

Random Color: green

Shuffled Colors: ['yellow', 'red', 'blue', 'green']

## 🧠 Difference Between math and random Modules

| Feature | math Module | random Module |
| --- | --- | --- |
| Purpose | Mathematical calculations | Generating random values |
| Type of Operations | Deterministic (always same result) | Non-deterministic (random result) |
| Example | math.sqrt(16) → 4.0 | random.randint(1,10) → varies each time |

## 🧩 Real-Life Examples

🎰 Example 1: Dice Roll Simulation

```
import random

dice = random.randint(1, 6)

print("You rolled a", dice)
```

🔢 Example 2: Generate OTP

```
import random
```

```
otp = random.randint(100000, 999999)
print("Your OTP is:", otp)
```

📈 Example 3: Area of Circle using math

```
import math
r = 7
area = math.pi * r * r
print("Area of Circle:", area)
```

## 🧩 Summary Table

| Module | Purpose | Common Functions |
|---|---|---|
| math | Perform mathematical operations | sqrt(), pow(), factorial(), floor(), ceil() |
| random | Generate random numbers or sequences | random(), randint(), choice(), shuffle() |

### 💡 Tip:

If you need **both math and random**, you can import them together:

```
import math, random
```

Then use:

```
print(math.sqrt(random.randint(1, 100)))
```

### ✅ Conclusion

- The math module is essential for **precise calculations** and **scientific tasks**.
- The random module helps in **games, sampling, OTPs, and simulations**.
- Together, they make Python powerful for **mathematics and probability-based applications**.