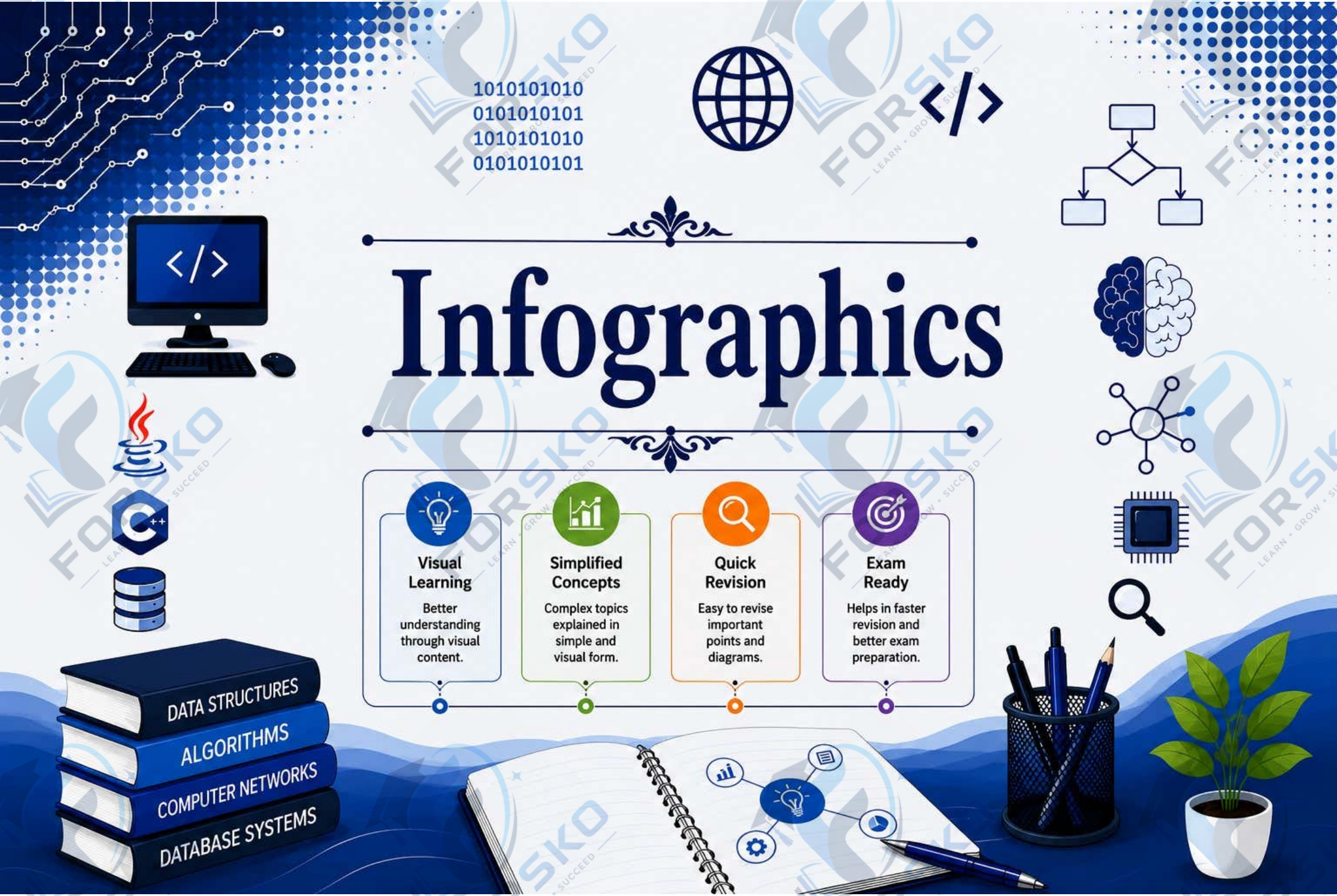

1010101010
0101010101
1010101010
0101010101
Infographics
Visual Learning
Better understanding through visual content.
Simplified Concepts
Complex topics explained in simple and visual form.
Quick Revision
Easy to revise important points and diagrams.
Exam Ready
Helps in faster revision and better exam preparation.
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS


1010101010
0101010101
1010101010
0101010101
Unit-I
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

# Introduction:
# History of Python Programming Language

Python is a high-level, interpreted, general-purpose programming language.
Its design philosophy emphasizes code readability with the use of significant indentation.

## Late 1980s
### The Beginning

Guido van Rossum, a programmer from the Netherlands, started working on Python.

## 1991
### First Release

Python 0.9.0 was released in February 1991. It included basic features like functions, modules, and exceptions.

## 1994
### Python 1.0


Python 1.0 was released in January 1994 with new features like lambda, map, filter, and reduce.

## 2000
### Python 2.0

Python 2.0 introduced new features such as list comprehensions and garbage collection improvements.

## 2008
### Python 3.0


Python 3.0 was released in December 2008 with major changes that made it not fully backward compatible.

## 2010s
### Rapid Growth

Python gained immense popularity in areas like Web Development, Data Science, AI, Automation, and more.

## 2020s and Beyond
### The Future

Python continues to evolve with new features, better performance, and a strong global community.

## Why Python Was Created?

- Guido wanted a language that was easy to read and write.
- Inspired by languages like ABC, Modula-3, and C.
- Aimed to reduce the complexity of programming.

## Key Highlights

- Named after the British comedy group "Monty Python".
- Open source and community-driven.
- Emphasizes readability and simplicity.
- *"Beautiful is better than ugly." – The Zen of Python*

## Today, Python is

- One of the most popular programming languages
- Used in countless industries and applications
- Loved by beginners and professionals alike

> *The mission of Python is to be a constructive force in the world.*
> – Guido van Rossum

python™

**Official Website**
https://www.python.org

# Thrust Areas of Python

Python is a versatile programming language that powers innovation across a wide range of domains.
Its simplicity, rich libraries, and strong community make it a top choice in today's world.

## 1 Web Development

- Build dynamic and scalable websites and web applications.
- Frameworks: Django, Flask, FastAPI
- Used by top companies like Instagram, Pinterest, YouTube

## 2 Data Science & Analytics

- Analyze, visualize, and extract insights from data.
- Libraries: NumPy, Pandas, Matplotlib, Seaborn
- Widely used in research, business intelligence, and decision making

## 3 Artificial Intelligence & Machine Learning

- Build intelligent systems that learn and improve from data.
- Libraries: TensorFlow, PyTorch, scikit-learn, Keras
- Applications in NLP, computer vision, robotics, and more

## 4 Automation & Scripting

- Automate repetitive tasks and streamline workflows.
- Used in system admin, file handling, web scraping, testing, and more.
- Boost productivity with simple and powerful scripts

## 5 Data Engineering & Big Data

- Build and manage large-scale data pipelines.
- Tools: PySpark, Airflow, Dask
- Handle massive data efficiently for analytics and ML applications

## 6 Scientific Computing & Research

- Perform complex mathematical computations and simulations.
- Libraries: SciPy, SymPy, NumPy
- Used in engineering, physics, chemistry, and academia

## 7 Desktop GUI Applications

- Build cross-platform desktop applications.
- Frameworks: Tkinter, PyQt, Kivy
- Suitable for tools, utilities, and enterprise applications

## 8 Game Development

- Create games using Python's simple syntax and libraries.
- Libraries/Engines: Pygame, Panda3D
- Great for 2D games, prototypes, and learning game logic

## 9 Internet of Things (IoT)

- Python connects hardware and software seamlessly.
- Libraries: MicroPython, RPi.GPIO, PySerial
- Used in smart devices, automation, sensors, and embedded systems

## 10 Education & Learning

- Beginner-friendly syntax makes it ideal for learning programming.
- Widely used in schools, colleges, and online courses.
- Builds a strong foundation for all tech fields

## Why Python Leads?

- ✓ Easy to learn and use
- ✓ Cross-platform and open source
- ✓ Huge ecosystem of libraries
- ✓ Strong community support

> *"Python empowers ideas, drives innovation, and builds the future."*

## Python Everywhere!

From startups to space missions, Python is the language of choice across industries and innovations.

python™

# Installing Anaconda Python Distribution

Anaconda is a powerful Python distribution for data science, machine learning, and scientific computing. It comes with Python, conda, and 250+ popular packages.


## Prerequisites

- 64-bit Windows / macOS / Linux
- Minimum 4 GB RAM (8 GB recommended)
- Internet connection

## Why Anaconda?

- Easy package management with conda
- Comes with Jupyter Notebook
- Cross-platform
- Great for Data Science & ML
- Large collection of pre-installed packages

## Download Anaconda

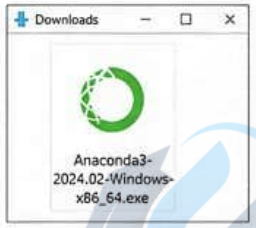
1. Go to the official website:
   https://www.anaconda.com/products/distribution
2. Choose your operating system
3. Click Download

Windows macOS Linux

Tip: Close all other applications before starting the installation.

## 1 Run the Installer

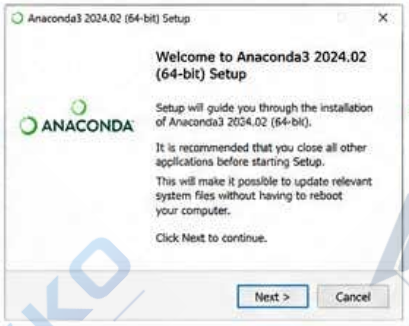
Locate the downloaded installer file and double-click to run it.




Windows example shown

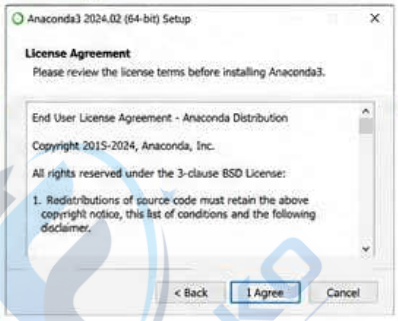
## 2 Welcome Screen

Click **Next** to proceed.



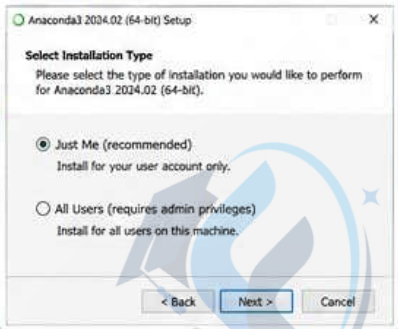

## 3 License Agreement

Read the license agreement and click **I Agree**.



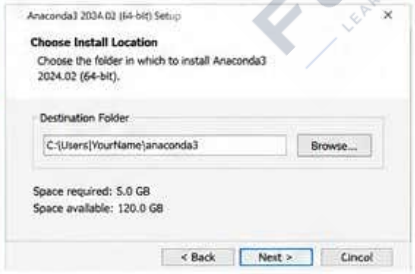

## 4 Choose Installation Type

Select **"Just Me (recommended)"** and click **Next**.



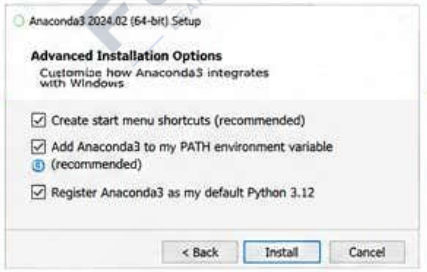

## 5 Choose Install Location

Select the destination folder or keep the default and click **Next**.



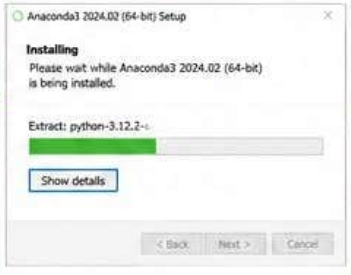

## 6 Advanced Installation Options

Select options as needed (recommended defaults are usually fine) and click **Install**.



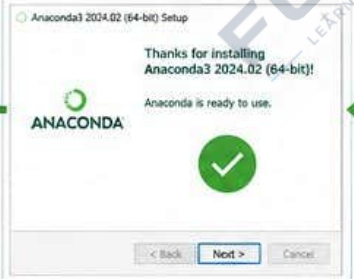

## 7 Installation Progress

Wait for the installation to complete.




## 8 Installation Complete

Click **Next** and then **Finish**.




## Verify Installation

Open **Command Prompt** / Terminal and type:

```
conda --version
```

You should see the conda version if installed successfully.


## Launch Anaconda Navigator

Search for "Anaconda Navigator" in Start Menu (Windows) or **Applications** (macOS) and open it.

You can manage environments, install packages, and run Jupyter Notebook from here.

## Get Started!

- Create environments with conda
- Install packages easily
- Launch Jupyter Notebook
- Start your Data Science journey!

Scan to learn more

★ Anaconda makes Python easy for everyone. Happy Coding!

# Installing PyCharm IDE
## to Setup a Python Development Environment

PyCharm is a powerful IDE by JetBrains for Python development.
It provides intelligent code assistance, debugging, testing, and integration with tools like Git, databases, and virtual environments.


### System Requirements

- Windows 10/11 / macOS / Linux
- 4 GB RAM (8 GB recommended)
- 2.5 GB free disk space
- Internet connection

### Why PyCharm?

- Smart code completion
- Powerful debugging tools
- Built-in Git integration
- Supports virtual environments
- Great for beginners & pros

### Download PyCharm

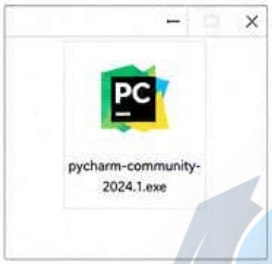
1. Go to the official website:
   https://www.jetbrains.com/pycharm/
2. Click on "Download"
3. Choose the edition:
   - Community (Free, open-source)
   - Professional (Paid, full features)
4. Select your OS and download

Windows macOS Linux

**Tip:** Community Edition is perfect for most Python developers.

### 1 Run the Installer

Locate the downloaded installer file and double-click to run it.

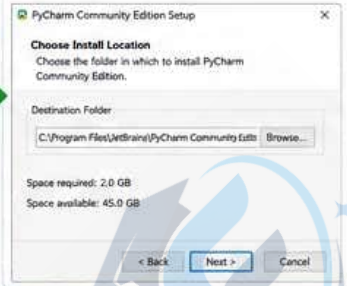



### 2 Welcome to PyCharm Setup

Click **Next** to continue.

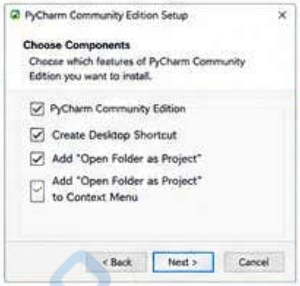

### 3 Choose Installation Location

Choose the folder where PyCharm will be installed and click **Next**.



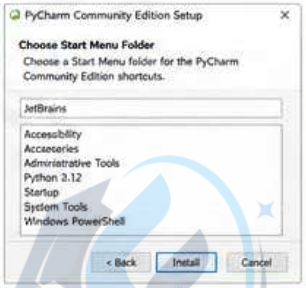

### 4 Choose Components

Select the components you want to install and click **Next**.




### 5 Start Menu Folder

Choose the Start Menu folder and click **Install**.

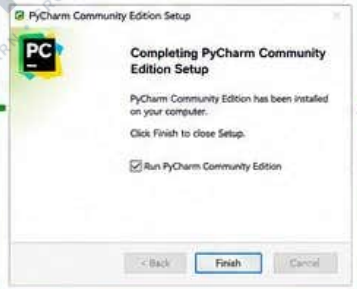



### 6 Installing

Wait while PyCharm is being installed on your system.


### 7 Installation Complete

Click **Finish** to close the installer.




### 8 Launch PyCharm

PyCharm will open after installation. You can also launch it from the desktop shortcut or Start Menu.


### 9 Set Up Environment (Optional)

You can configure your IDE.
Recommended:

- Choose a theme
- Install useful plugins
- Configure Python interpreter


### 10 You're All Set!

Your Python development environment is ready.

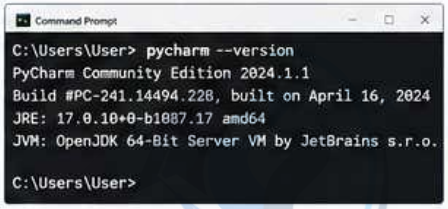
Happy Coding!

### Verify Installation

Open Terminal / Command Prompt and type:

```
pycharm --version
```

You should see the installed PyCharm version.




### Useful First Steps in PyCharm

- Create a new project
- Create a Python file and run your first program
- Use built-in terminal
- Explore code completion and shortcuts (Ctrl + Space)
- Learn more: Help > Learn IDE Features

### Helpful Tips

- Keep PyCharm and plugins updated
- Use virtual environments (venv / conda)
- Explore shortcuts: File > Settings > Keymap
- Leverage version control (Git) for your projects

Enjoy smart coding with PyCharm!

PyCharm + Python = Powerful Productivity

Code Better. Debug Faster. Build Smarter.

# Creating and Running Your First Python Project

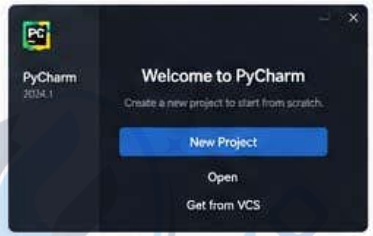
Let's create a simple "Hello, World!" Python project using PyCharm IDE and run it step by step.


## What You'll Learn

- Create a new project
- Write a Python program
- Run your program
- See the output

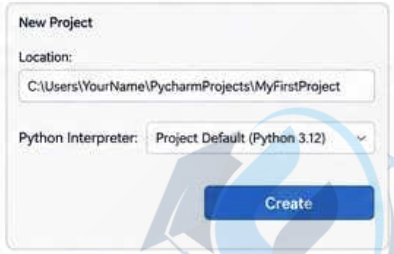
## 1 Open PyCharm

Launch PyCharm IDE.
On the Welcome screen, click "New Project".



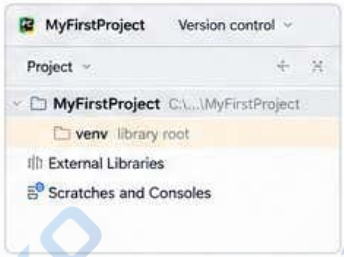

## 2 Create New Project

Choose a location for your project, give it a name, and click **Create**.



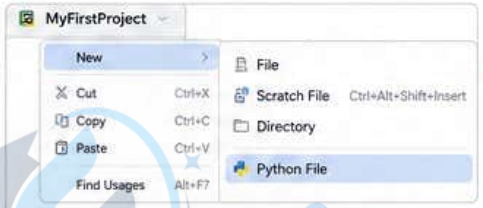

## 3 Project Created

Your new project is now ready in PyCharm.



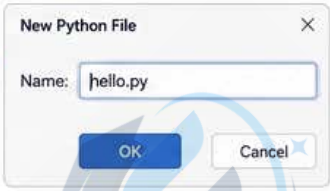

## 4 Create a Python File

Right-click on the project folder (MyFirstProject) or the main folder → **New** → **Python File**.




## 5 Name Your File

Name the file **hello.py** and click **OK**.




## 6 Write Your Code

Type the following code in **hello.py**:

```
hello.py
1  # My first Python project
2
3  print("Hello, World!")
```

**Tip:** Python is case-sensitive. Make sure the syntax is correct.

## 7 Run Your Program

Right-click anywhere in the editor and select **Run 'hello'** or click the green ▶ button on the top-right.


## 8 Program is Running

PyCharm will run your program. Wait for the output.


## 9 See the Output

The output will appear in the Run tool window.


## 10 Congratulations!

You have successfully created and run your first Python project!

**Keep coding and keep learning!**

## Project Structure

```
MyFirstProject/
    hello.py
    venv/      (virtual environment)
External Libraries
Scratches and Consoles
```

## What is Happening?

1. You create a project (a folder) to keep your files organized.
2. You write Python code in a file with .py extension.
3. You run the file using Python interpreter.
4. The output is displayed in the Run window.

## Common Next Steps

- Add more print() statements
- Try taking input from the user
- Create functions
- Build something amazing!

## Remember

- ☑ Save your file before running.
- ☑ File extension must be .py
- ☑ Check the output and fix errors if any.

**You just wrote and ran your first Python program. This is the first step of your amazing Python journey!**

# Parts of Python Programming Language: Identifiers, Keywords

Like every programming language, Python has building blocks.
Two important building blocks are **Identifiers** and **Keywords**.

## IDENTIFIERS

**Identifiers** are names given to entities like variables, functions, classes, objects, modules, etc.

### Rules for Identifiers

- Must start with a letter (A–Z or a–z) or underscore (_).
- After the first character, you can use letters, digits (0–9) or underscores (_).
- Identifiers are case-sensitive (name, Name and NAME are different).
- Cannot be a Python keyword.
- No special characters allowed (except _).

### Examples

**Valid Identifiers**

| | | |
|---|---|---|
| `total` | `_count` | `value2` |
| `student_name` | `getData` | `num_1` |
| `A1` | `_temp` | `myVariable` |

**Invalid Identifiers**

| | | |
|---|---|---|
| `2value` | `student-name` | `total$` |
| `class` | `my var` | `@data` |
| `if` | `temp!` | `1st` |

### Example in Code

```
# Valid identifiers
student_name = "Alice"
total_marks = 450
_count = 1
print(student_name, total_marks, _count)
```

### Good Practice

Use meaningful and descriptive names so your code is easy to read and understand.

## KEYWORDS

**Keywords** are reserved words in Python that have a special meaning.

### Characteristics

- Keywords are part of Python syntax.
- They cannot be used as identifiers (variable names, function names, etc.).
- All keywords are written in lowercase.

### List of Python Keywords (Python 3.x)

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

### Example (Invalid Use)

```
# Invalid: using keyword as variable name
if = 10          # ✗ SyntaxError
for = 5          # ✗ SyntaxError
```

**Note:** You cannot use keywords as names for variables, functions, classes, or any other identifiers.

### Check All Keywords

You can get the list of all keywords in Python using the keyword module.

```
import keyword
print(keyword.kwlist)
```

### Identifiers vs Keywords

| Feature | Identifiers | Keywords |
|---|---|---|
| Purpose | User-defined names | Reserved by Python |
| Examples | `name, total, value1` | `if, for, while` |
| Can we change? | Yes | No |
| Can be used as names? | Yes | No |

### Tips

- Choose meaningful identifiers.
- Follow naming conventions: use lowercase with underscores (e.g., `total_marks`).
- Avoid using Python keywords as identifiers.

### Quick Check

Which of the following are valid identifiers?

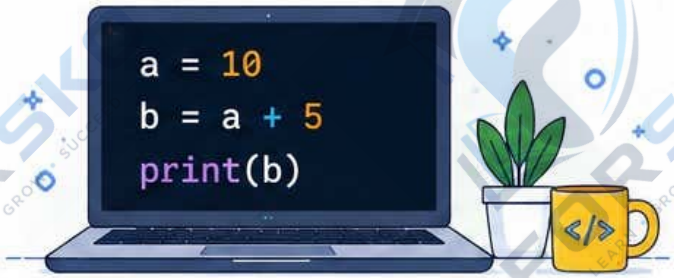
- ✓ student1
- ✓ _value
- ✓ total_marks
- ✗ 2name
- ✗ for
- ✗ my-name

Learn the rules, follow the style, write better Python programs!

**Identifiers are your names. Keywords are Python's words. Use them correctly to write powerful programs!**

# Python Basics: Statements and Expressions

Every Python program is made up of **statements**.
Statements are built from **expressions**.




## STATEMENTS

A statement is a complete instruction that Python can execute. It **performs an action**.

### Examples of Statements

| Statement Type | Example | What it does |
|---|---|---|
| Assignment Statement | `x = 25` | Assigns the value 25 to variable x |
| Print Statement | `print(x)` | Displays the value of x |
| Input Statement | `name = input("Enter name: ")` | Takes input from the user |
| Conditional Statement | `if x > 0:`<br>`  print("Positive")` | Executes block if condition is true |
| Loop Statement | `for i in range(3):`<br>`  print(i)` | Repeats a block of code |
| Function Call Statement | `greet("Hello")` | Calls a function |

**Key Point:** A statement stands on its own as a complete instruction.

## EXPRESSIONS

An expression is a combination of values, variables, operators, and function calls that **evaluates to a value.**

### Examples of Expressions

| Expression | Evaluates to |
|---|---|
| `10` | 10 |
| `x` | value of x |
| `x + 5` | sum of x and 5 |
| `(a * b) / 2` | result of a * b, then divided by 2 |
| `len("Python")` | 6 |
| `x > 0` | True or False |

**Key Point:** An expression always produces a value.

### RELATIONSHIP

**Expressions** Produce a value → **Used in** → **Statements** Perform an action

**Example**

```
x = 10 + 5      # 10 + 5 is an expression that evaluates to 15
print(x)        # print(x) is a statement that displays the value
```

**More Examples**

```
total = price * qty      # price * qty → expression
if total > 100:          # total > 100 → expression used in a statement
    print("Discount")    # print(...) → statement
```

## DETAILED EXAMPLES

### 1. Assignment Statement

```
a = 7 + 3     # 7 + 3 is an expression (evaluates to 10)
b = a * 2     # a * 2 is an expression (evaluates to 20)
print(b)      # print(b) is a statement
```

### 2. Conditional Statement

```
x = 15                  # statement
if x % 2 == 0:          # condition is an expression
    print("Even")       # statement
else:
    print("Odd")        # statement
```

### 3. Loop Statement

```
for i in range(3):      # statement
    print(i)            # print(i) is a statement
```

### 4. Function Call Statement

```
def add(a, b):          # def is a statement
    return a + b        # a + b is an expression

result = add(4, 6)      # function call statement
print(result)           # print is a statement
```

## COMMON MISTAKES

**Mistake 1: Writing an expression by itself**

```
10 + 5          # This is an expression, but not a statement
                # Nothing will be displayed or stored
```

**Mistake 2: Forgetting indentation in block statements**

```
if x > 0:
print("Positive")     # IndentationError
```

**Mistake 3: Using keywords as identifiers**

```
for = 10              # SyntaxError: invalid syntax
```

## QUICK SUMMARY

- Expressions evaluate to a value.
- Statements perform an action.
- Expressions are used inside statements.
- A program is a sequence of statements.

**Think of it this way:**

Expression `2 + 3` (Produces a value) → **Used in** → Statement `x = 2 + 3` (Performs an action)

**Remember:** Expressions give values; Statements do something with those values.

Master statements and expressions – build the foundation of every Python program!

# Python Basics: Variables, Operators

Variables are used to store data. Operators are used to perform operations on data and variables.


## VARIABLES

A variable is a name (identifier) that refers to a value stored in computer memory. The value can change during program execution.

### 1. Creating Variables

- No need to declare type in Python.
- Use meaningful names.
- Case-sensitive.

Examples:

```
x = 10
name = "Alice"
pi = 3.14
is_valid = True
```

### 2. Rules for Variables

- Must start with a letter (A–Z or a–z) or underscore (_).
- After the first character, can contain letters, digits (0–9) or underscores (_).
- Cannot be a Python keyword.
- Case-sensitive (**age**, **Age** and AGE are different).
- No spaces or special characters allowed.

### 3. Updating Variables

The value stored in a variable can be changed.

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

### 4. Multiple Assignment

You can assign multiple variables in one statement.

```
a, b, c = 1, 2, 3
x = y = z = 0
```

### 5. Data Types (Examples)

| Type | Example | Description |
|---|---|---|
| int | x = 10 | Integer (whole number) |
| float | p = 3.14 | Decimal number |
| str | name = "Bob" | Text (string) |
| bool | flag = True | Boolean (True or False) |

### Tips

- Choose meaningful variable names.
- Follow **snake_case** style (e.g., **total_marks**).
- Avoid using keywords as variable names.

### Common Mistakes

```
1variable = 10      # Invalid: cannot start with digit
total marks = 100   # Invalid: spaces not allowed
for = 5             # Invalid: 'for' is a keyword
```

## OPERATORS

Operators are special symbols that perform operations on one, two or more operands (values/variables).

### TYPES OF OPERATORS

### 1. Arithmetic Operators

| Operator | Meaning | Example | Result |
|---|---|---|---|
| + | Addition | 7 + 3 | 10 |
| - | Subtraction | 7 - 3 | 4 |
| * | Multiplication | 7 * 3 | 21 |
| / | Division (float) | 7 / 3 | 2.333... |
| // | Floor Division | 7 // 3 | 2 |
| % | Modulus (Remainder) | 7 % 3 | 1 |
| ** | Exponentiation | 7 ** 3 | 343 |

### 2. Comparison (Relational) Operators

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

### 3. Assignment Operators

| Operator | Meaning | Example | Same as |
|---|---|---|---|
| = | Assign | x = 10 | x = 10 |
| += | Add and assign | x += 5 | x = x + 5 |
| -= | Subtract and assign | x -= 3 | x = x - 3 |
| *= | Multiply and assign | x *= 2 | x = x * 2 |
| /= | Divide and assign | x /= 4 | x = x / 4 |
| %= | Modulus and assign | x %= 3 | x = x % 3 |
| **= | Exponent and assign | x **= 2 | x = x ** 2 |
| //= | Floor divide and assign | x //= 2 | x = x // 2 |

### 4. Logical Operators

| Operator | Meaning | Example | Result |
|---|---|---|---|
| and | True if both are True | (5 > 3) and (2 < 4) | True |
| or | True if at least one is True | (5 > 3) or (2 > 4) | True |
| not | Reverse the result | not (5 > 3) | False |

### 5. Membership Operators

| Operator | Meaning | Example | Result |
|---|---|---|---|
| in | True if value exists in sequence | 'a' in 'apple' | True |
| not in | True if value does not exist in sequence | 'z' not in 'apple' | True |

### 6. Identity Operators

| Operator | Meaning | Example | Result |
|---|---|---|---|
| is | True if both operands refer to same object | a is b | True/False |
| is not | True if both operands refer to different objects | a is not b | True/False |

### Operator Precedence (Highest to Lowest)

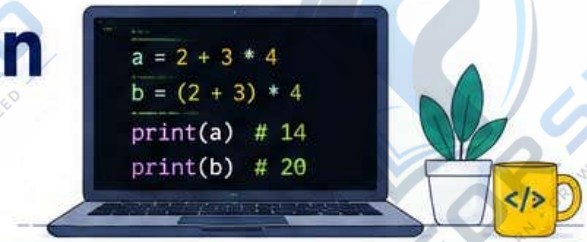
() → ** → unary +, not → *, /, //, % → +,- → comparison operators → not → and → or

Use parentheses () to change the order.

Variables store data. Operators work on data. Together they build powerful Python programs!

# Precedence and Associativity in Python

When an expression contains multiple operators, the order in which they are evaluated is determined by **Precedence** and **Associativity**.




## PRECEDENCE

**Precedence** decides which operator is evaluated first in an expression. Operators with higher precedence are evaluated before operators with lower precedence.

## ASSOCIATIVITY

Associativity decides the order of evaluation when two or more operators of the same precedence appear in an expression. It can be Left-to-Right or Right-to-Left.

### Operator Precedence in Python (Highest to Lowest)

| Precedence Level | Operators | Description | Example | Evaluates As |
|---|---|---|---|---|
| 1 | ( ) | Parentheses (grouping) | (2 + 3) * 4 | 5 * 4 = 20 |
| 2 | ** | Exponentiation | 2 ** 3 ** 2 | 2 ** (3 ** 2) = 512 |
| 3 | +x -x ~x | Unary plus, minus, bitwise NOT | -5 + 2 | (-5) + 2 = -3 |
| 4 | * / // % | Multiplication, Division, Floor Division, Modulus | 7 * 3 / 2 | 21 / 2 = 10.5 |
| 5 | + - | Addition, Subtraction | 7 + 3 - 2 | 10 - 2 = 8 |
| 6 | << >> | Bitwise Left shift, Right shift | 8 << 1 | 16 |
| 7 | & | Bitwise AND | 5 & 3 | 1 |
| 8 | ^ | Bitwise XOR | 5 ^ 3 | 6 |
| 9 | \| | Bitwise OR | 5 \| 3 | 7 |
| 10 | == != > < >= <= | Comparisons | 5 > 3 | True |
| 11 | not | Logical NOT | not True | False |
| 12 | and | Logical AND | True and False | False |
| 13 | or | Logical OR | True or False | True |
| 14 | if - else | Conditional (ternary) expression | x if c else y | Depends on c |
| 15 | = += -= *= /= //= %= **= &= \|= ^= <<= >>= | Assignment operators | x += 1 | x = x + 1 |

**Note:** If operators have different precedence, the one with higher precedence is evaluated first. If they have the same precedence, associativity decides the order.

### Associativity in Python

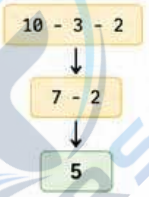
#### Left-to-Right (Most operators)

When operators of the same precedence appear, evaluation is done from left to right.

Examples: +, -, *, /, //, %, &, |, ^, ==, !=, <, >, <=, >=, and, or

**Example 1:**

```
10 - 3 - 2
=(10 - 3) - 2
= 7 - 2
= 5
```



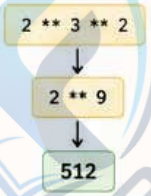

#### Right-to-Left (Few operators)

When operators of the same precedence appear, evaluation is done from right to left.

Examples: **, = (and other assignment operators)

**Example 2 (Right-to-Left for **):**

```
2 ** 3 ** 2
= 2 ** (3 ** 2)
= 2 ** 9
= 512
```

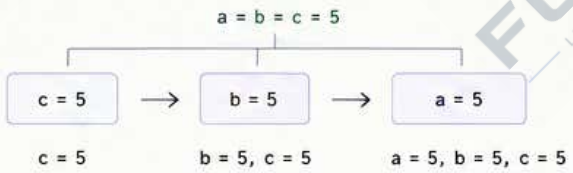



**Example 3 (Assignment – Right-to-Left)**

```
a = b = c = 5
```

Evaluates as: a = (b = (c = 5))

First c = 5, then b = 5, then a = 5.




### Let's Evaluate Some Expressions

**Example A**

```
expr = 2 + 3 * 4 - 5 / (1 + 1)
Step 1: ()
    2 + 3 * 4 - 5 / 2
Step 2: * and /
    2 + 12 - 2.5
Step 3: + and - (Left to Right)
    14 - 2.5 = 11.5
```

Result: 11.5

**Example B**

```
expr = 2 ** 3 ** 2 + 10
Step 1: ** (Right to Left)
    2 ** (3 ** 2) + 10
    = 2 ** 9 + 10
Step 2: +
    = 512 + 10 = 522
```

Result: 522

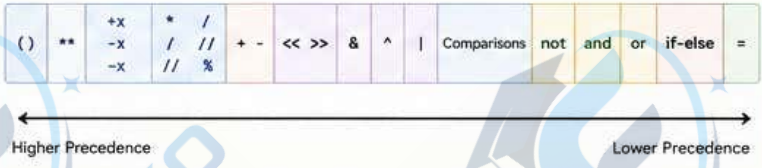
### Key Takeaways

- Parentheses () have the highest precedence.
- Exponentiation ** is right-associative.
- Most operators are left-associative.
- When in doubt, use parentheses to make the order clear.

### Quick Tips

- Operators with higher precedence are evaluated first.
- If precedence is same, associativity decides.
- Use parentheses to avoid confusion and improve readability.

### Precedence Order (Highest → Lowest)



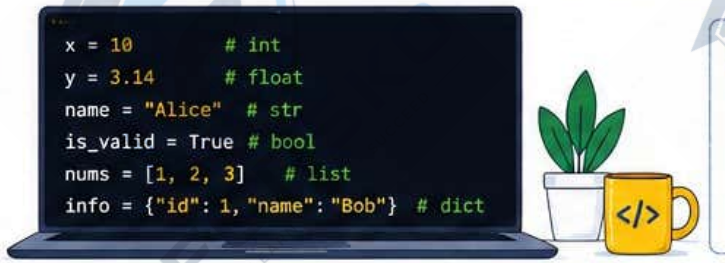

Understanding Precedence and Associativity helps you predict how Python evaluates expressions accurately.

Write clear expressions. Write better Python!


1010101010
0101010101
1010101010
0101010101
Unit -II
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

# Data Types in Python

A data type specifies the type of value a variable can hold and the operations that can be performed on that value.




**Remember**

The type of data determines what values can be stored and what operations can be performed.

## Built-in Data Types in Python

### 1 Numeric (int, float, complex)

Used to store numbers.

- `int` → Whole numbers
- `float` → Decimal numbers
- `complex` → Complex numbers

**Examples**

```
a = 10          # int
b = 3.14        # float
c = 2 + 3j      # complex
```

**Type: int, float, complex**

**Properties**

- int has unlimited precision.
- float follows double precision.
- complex is written as `a + bj`.

### 2 String (str)

Used to store sequence of characters (text).

**Examples**

```
s1 = "Hello"
s2 = 'Python'
s3 = "123"
```

**Type: str**

**Properties**

- Enclosed in single, double or triple quotes.
- Immutable (cannot be changed).
- Supports indexing and slicing.

### 3 Boolean (bool)

Used to store truth values.

**Examples**

```
b1 = True
b2 = False
```

**Type: bool**

**Properties**

- Two values only: `True` or `False`.
- Used in conditions and logical operations.

### 4 List (list)

Used to store ordered collection of items. Mutable (can change).

**Examples**

```
nums = [1, 2, 3, 4]
mix = [10, "Hi", 3.5, True]
empty = []
```

**Type: list**

**Properties**

- Ordered, mutable.
- Allows duplicate values.
- Supports indexing, slicing, `append()`, `remove()`, etc.

### 5 Tuple (tuple)

Used to store ordered collection of items. Immutable (cannot change).

**Examples**

```
t1 = (1, 2, 3)
t2 = (10, "Hi", 3.5)
empty = ()
```

**Type: tuple**

**Properties**

- Ordered, immutable.
- Allows duplicate values.
- Supports indexing and slicing.
- Faster and safer than lists.

### 6 Dictionary (dict)

Used to store key-value pairs. Mutable.

**Examples**

```
student = {"id": 1,
           "name": "Alice",
           "marks": 95}
```

**Type: dict**

**Properties**

- Unordered (in versions < 3.7).
- Keys must be unique and immutable (int, str, tuple, etc.).
- Access values using keys.

### 7 Check Data Type

Use the `type()` function.

```
type(10)          # <class 'int'>
type(3.14)        # <class 'float'>
type("Hello")     # <class 'str'>
type(True)        # <class 'bool'>
type([1,2,3])     # <class 'list'>
type((1,2,3))     # <class 'tuple'>
type({"a"::1))    # <class 'dict'>
```

### 8 Type Casting (Conversion)

Convert one data type to another using built-in functions.

| Function | Description | Example |
|---|---|---|
| `int(x)` | Converts x to integer | `int(3.9)` → `3` |
| `float(x)` | Converts x to float | `float("2.5")` → `2.5` |
| `str(x)` | Converts x to string | `str(100)` → `'100'` |
| `list(x)` | Converts x (iterable) to list | `list((1,2))` → `[1, 2]` |
| `tuple(x)` | Converts x (iterable) to tuple | `tuple([1,2])` → `(1, 2)` |
| `dict(x)` | Converts key-value iterable to dict | `dict([("a",1)])` → `{'a': 1}` |
| `bool(x)` | Converts x to boolean | `bool(0)` → `False`, `bool(5)` → `True` |

### Common Mutable vs Immutable Types

**Mutable (can be changed)**

list, dict, set

**Immutable (cannot be changed)**

int, float, bool, str, tuple, complex

Trying to change an immutable object creates a new object.

### Key Points

- Choosing the correct data type makes your program efficient and bug-free.
- Use list for changeable sequences and tuple for fixed sequences.
- Use dict to store data in key-value pairs for fast access.
- Python is dynamically typed – you do not need to declare the type of a variable.

Understanding data types is the foundation of writing correct and efficient Python programs.

# Indentation and Comments in Python

Python uses indentation to define blocks of code.
Comments are used to explain code and make it more readable.


**Remember**
Good indentation avoids errors.
Good comments improve clarity.

## 1. INDENTATION

Indentation is the spacing at the beginning of a line.
It tells Python which statements belong to a block.

### Rules of Indentation

- Use consistent indentation throughout the program. (Recommended: 4 spaces per indentation level)
- All statements inside a block must have the same indentation level.
- A new block starts after a colon (:)
- Python does not allow mixing spaces and tabs. Choose one and be consistent.

### Example: Correct Indentation

```
1  x = 10
2  if x > 5:
3      print("x is greater than 5")        → Block 1
4  else:
5      print("x is 5 or less")            → Block 2
6  print("This is outside the if block") → Outside blocks
```

### Example: Incorrect Indentation ( causes IndentationError )

```
1  x = 10
2  if x > 5:
3      print("x is greater than 5")
4  else:
5      print("x is 5 or less")
6  print("This will cause an error")
```

**✖ Problem**
The `print()` statements are not indented correctly.
Python will show:
`IndentationError`

### Common Indentation Situations

| Situation | What Happens |
|---|---|
| Missing indentation after : | IndentationError |
| Extra indentation without reason | IndentationError |
| Inconsistent use of spaces and tabs | TabError |
| Correct indentation | Program runs fine |

### Best Practices

- Use 4 spaces per indentation level (most common standard).
- Do not mix tabs and spaces.
- Use meaningful block structure to improve readability.
- Use your editor settings to show indentation guides.

### Visualizing Blocks

```
1  if True:                          → Level 0
2      print("Level 1")              → Level 1
3      if True:
4          print("Level 2")          → Level 2
5      print("Back to Level 1")      → Back to Level 1
6  print("Level 0")                  → Back to Level 0
```

## 2. COMMENTS

Comments are ignored by Python interpreter.
They are for humans, not for computers!

### Types of Comments

#### 1. Single-line Comments

Start with #. The rest of the line is ignored.

```
x = 10          # This is a comment
y = 20          # Another comment
print(x + y)    # Add x and y
```

#### 2. Multi-line Comments

Use triple quotes ''' or """.
Everything inside is ignored.

```
'''
This is a multi-line comment.
It can span multiple lines.
Useful for long explanations.
'''
print("Hello!")
```

### Where to Use Comments?

- To explain what the code does
- To explain why a particular approach is used
- To leave notes for yourself or others
- To temporarily disable code (use with caution)

### Example with Comments

```
1  # Program to calculate area of rectangle
2  length = 10       # length of the rectangle
3  width = 5         # width of the rectangle
4
5  # Calculate area
6  area = length * width      # area = l * w
7
8  print("Area =", area)      # display the result
```

### Common Mistakes

- Using comments to state obvious things:
  `x = 5  # x is 5` (Not useful)
- Outdated comments (not matching code):
  `y = 10  # y stores name` (Misleading)
- Too many comments (makes code noisy):
  `a = b + c  # a equals b plus c` (Unnecessary)

### Key Takeaways

- Indentation defines block structure in Python.
- Use consistent indentation (prefer 4 spaces).
- Comments make code readable and maintainable.
- Good code = Clear structure + Helpful comments.

**Pro Tip:** Write code for machines to run, but write it well for humans to read!

**Clean indentation + Meaningful comments = Readable, Maintainable, and Professional Python Code!**

# Reading Input, Printing Output in Python

Programs interact with users through Input and Output.
Python provides simple functions to read input from the user and display output on the screen.


## 1. READING INPUT

Use the `input()` function to read data from the user.
It always returns data as a string.

### Syntax

```
variable = input("Prompt message")
```

### How it works

- The **prompt message** is displayed to the user.
- The user types something and presses Enter.
- The typed data is stored as a string in the variable.

### Example 1: Simple Input

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

**Run:**

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

### Example 2: Input with Type Conversion

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

**Run:**

```
Enter your age: 20
Enter your height (in m): 1.72
Age: 20 years
Height: 1.72 meters
```

### Important Points

- `input()` returns data as a string. Convert it to `int()`, `float()`, or other types when needed.
- Use meaningful prompts to guide the user.
- The program waits for user input at the `input()` line.

### Common Mistakes

- Forgetting type conversion:
  ```
  age = input("Enter age: ") + 5    # Error! (string + int)
  ```
- Using `input()` without a prompt:
  ```
  name = input()                    # No message for user
  ```
- Expecting numbers but not converting:
  ```
  x = input("Enter a number: ") * 2  # Repeats string, not multiply!
  ```

### Quick Tips

- Always convert input to the required type.
- Validate user input when necessary.
- Use clear prompts and messages.

## 2. PRINTING OUTPUT

Use the `print()` function to display output.
It can print text, variables and expressions.

### Syntax

```
print(value1, value2, ..., sep=' ', end='\n')
```

### How it works

- By default, items are separated by a space (`sep=' '`).
- By default, `print()` moves to a new line after printing (`end='\n'`).
- You can change `sep` and `end` for customized output.

### Commonly Used Parameters

| Parameter | Default | Description | Example |
|---|---|---|---|
| sep | ' ' (space) | Separator between items | `print(1, 2, 3, sep=", ")` Output: 1, 2, 3 |
| end | '\n' (new line) | What to print at the end | `print("Hello", end="! ")` `print("Python")` Output: Hello! Python |
| file | sys.stdout | Where to print | (Advanced topic) |
| flush | False | Force immediate output | (Advanced topic) |

### Examples

**1. Printing Text**

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

Output:
```
Hello, World!
```

**2. Printing Variables and Expressions**

```
x = 10
y = 5
print("Sum =", x + y)
```

Output:
```
Sum = 15
```

**3. Multiple Items**

```
name = "Alice"
age = 20
print("Name:", name, "Age:", age)
```

Output:
```
Name: Alice Age: 20
```

**4. Using sep and end**

```
print("A", end=" ")
print("B", end=" ")
print("C")
```

Output:
```
A B C
```

### Example: Complete Program

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Hello,", name + "!")
print("You will be", age + 1, "next year.")
```

**Sample Run**

```
Enter your name: Bob
Enter your age: 19
Hello, Bob!
You will be 20 next year.
```

### Key Takeaways

- `input()` is used to read data from the user.
- `print()` is used to display output.
- Input helps programs become interactive.
- Clear output makes programs user-friendly.

**Input** takes data from the user. **Output** shows results to the user.
Together they make Python programs **interactive** and useful!

**Pro Tip:** Test your program with different inputs to make it robust!

# Type Conversions in Python

Type conversion (or type casting) is the process of converting one data type into another. It helps in performing operations between different types of data.

```
x = 10        # int
y = 3.14      # float
s = '25'      # str

a = int(s)    # str to int
b = float(x)  # int to float
c = str(y)    # float to str
```

### Key Idea

Convert data to the right type whenever required. Wrong type can cause errors or unexpected results.

## 1. Why Type Conversion?

- User input is always taken as string.
- Operations between different types may not be allowed.
- Converting to appropriate type avoids errors and ensures correct results.

**Example:**

```
a = input("Enter a number: ")   # string
b = a + 5          # Error
b = int(a) + 5     # Works
```

```
TypeError: can only concatenate str
(not "int") to str
```

## 2. Built-in Conversion Functions in Python

| Function | Converts To | Example | Result |
|---|---|---|---|
| `int(x)` | Integer | `int('25')`<br>`int(3.9)`<br>`int(True)` | 25<br>3<br>1 |
| `float(x)` | Floating point | `float('3.14')`<br>`float(10)`<br>`float(True)` | 3.14<br>10.0<br>1.0 |
| `str(x)` | String | `str(123)`<br>`str(3.14)`<br>`str(True)` | '123'<br>'3.14'<br>'True' |
| `bool(x)` | Boolean | `bool(0)`<br>`bool(25)`<br>`bool('')`<br>`bool('Hello')` | False<br>True<br>False<br>True |

## 3. Common Conversions with Examples

| Conversion | Code | Output | Explanation |
|---|---|---|---|
| String to int | `x = int('42')`<br>`print(x)` | 42 | Numeric string to integer |
| String to float | `y = float('3.75')`<br>`print(y)` | 3.75 | Numeric string to float |
| Int to float | `z = float(10)`<br>`print(z)` | 10.0 | Integer to floating point |
| Float to int | `a = int(7.99)`<br>`print(a)` | 7 | Decimal part is truncated |
| Int to string | `b = str(100)`<br>`print(b)` | '100' | Integer to string |
| Float to string | `c = str(3.5)`<br>`print(c)` | '3.5' | Float to string |
| Bool conversions | `print(bool(0))`<br>`print(bool(1))`<br>`print(bool(''))`<br>`print(bool('Hi'))` | False<br>True<br>False<br>True | Zero or empty → False<br>Non-zero or non-empty → True |

## 4. User Input and Type Conversion

`input()` always returns data as string. Convert it to required type.

```
age = int(input("Enter your age: "))    # convert to int
price = float(input("Enter price: "))   # convert to float
name = input("Enter your name: ")       # already string
```

**Sample Run**

```
Enter your age: 19
Enter price: 99.95
Enter your name: Alice
```

→

**Stored As**

```
age   → 19      (int)
price → 99.95   (float)
name  → 'Alice' (str)
```

## 5. Decimal to Integer Conversion

`int()` removes the fractional part (truncates toward zero).

```
print(int(9.7))     # 9
print(int(-9.7))    # -9
```

Note: Use `round()`, `floor()`, `ceil()` for other needs.

| Function | Meaning | Example (x = 3.6) | Output |
|---|---|---|---|
| `round(x)` | Rounds to nearest | `round(x)` | 4 |
| `math.floor(x)` | Rounds down | `floor(x)` | 3 |
| `math.ceil(x)` | Rounds up | `ceil(x)` | 4 |

## 6. Important Points

- Choose the correct type for your data.
- Always validate and convert user input.
- Invalid conversions raise ValueError.

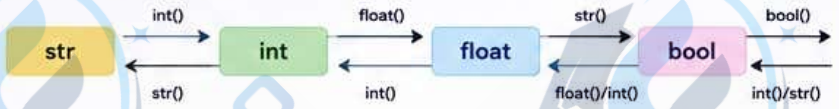
**Example of Error**

```
x = int('abc')    # invalid literal
# ValueError: invalid literal for int() with base 10: 'abc'
```

**Safe Conversion Using try-except**

```
try:
    x = int(input("Enter a number: "))
    print("You entered:", x)
except ValueError:
    print("Invalid input! Please enter a valid number.")
```

## 7. Conversion Summary




**Invalid Conversions (Examples)**

- `int('12.5')` # ValueError
- `float('abc')` # ValueError
- `int(True.5)` # ValueError

**Right type, right result! Use type conversion smartly to write error-free and efficient Python programs.**

# The type() Function and Is Operator in Python


Python provides built-in tools to inspect data and compare objects.
The type() function tells us the data type of a value, and the is operator checks object identity.

## 1. The type() Function

The type() function returns the type (class) of the object passed as an argument.

**Syntax**

```
type(object)
```

**How it works**

- It returns the class of the object as a type object.
- The output is of type 'type'.

**Examples**

```
type(10)          # <class 'int'>
type(3.14)        # <class 'float'>
type('Hello')     # <class 'str'>
type([1, 2, 3])   # <class 'list'>
type((1, 2))      # <class 'tuple'>
type(True)        # <class 'bool'>
type({'a':1})     # <class 'dict'>
type(None)        # <class 'NoneType'>
```

**Common Return Types**

| Data Type | Example | Data Type | Example |
|---|---|---|---|
| int | type(5) → <class 'int'> | tuple | type((1,2)) → <class 'tuple'> |
| float | type(3.5) → <class 'float'> | dict | type({'a':1}) → <class 'dict'> |
| str | type('hi') → <class 'str'> | bool | type(False) → <class 'bool'> |
| list | type([1,2]) → <class 'list'> | NoneType | type(None) → <class 'NoneType'> |

**Practical Use**

- To check the type of data stored in a variable.
- To write type-safe programs.
- To perform different actions based on data type.

**Example**

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

print(type(x))          # Always <class 'str'>
n = int(x)              # Convert to int
print(type(n))          # Now <class 'int'>
```

**Keep in Mind**

- type(1) and type(True) are different: <class 'int'> vs <class 'bool'>
- None has its own type: <class 'NoneType'>

## 2. The is Operator

The is operator checks whether two variables point to the same object in memory (identity), not just equal values.

**Syntax**

```
object1 is object2
```

**How it works**

- Returns True if both refer to the same object.
- Returns False if they refer to different objects, even if values are equal.

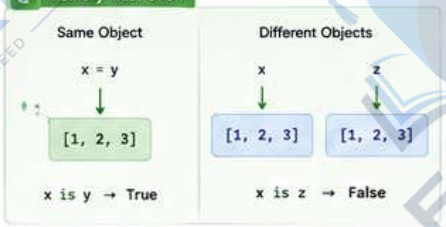
**Examples**

```
a = 10
b = 10
print(a is b)           # True (same object)

x = [1, 2, 3]
y = x
z = [1, 2, 3]
print(x is y)           # True  (same object)
print(x is z)           # False (different objects)
```

**is vs == (Important Difference)**

| Operator | What it checks | Example | Output |
|---|---|---|---|
| == | Value Equality (values are equal) | a = [1,2]<br>b = [1,2]<br>print(a == b) | True |
| is | Identity Equality (same object in memory) | a = [1,2]<br>b = [1,2]<br>print(a is b) | False |

**Memory Illustration**




**Common Use Cases**

- Check if a variable is None: if value is None:
- Check for singletons like True, False, None.
- Useful with immutable singletons.

**Examples**

```
user_input = None
if user_input is None:
    print("No input provided")

if flag is True:
    print("Flag is True")
```

**Common Mistakes**

- Using is to compare values → may give unexpected results.
- Do not use 'is' with numbers, strings, or lists for value comparison. Use '==' instead.

Wrong ✖

```
a = 1000
b = 1000
print(a is b)  # May be False
```

Right ✔

```
a = 1000
b = 1000
print(a == b)  # True
```

**Key Takeaway:** Use type() to know what a value is.
Use is to check whether two variables are the exact same object.

**Pro Tip:** When in doubt:
Use '==' for value comparison, and 'is' for identity comparison.

# Dynamic and Strongly Typed Language in Python


Python is a **Dynamic** and **Strongly Typed** language.
This combination gives flexibility to the programmer while ensuring safety and reliability in code.

## 1. DYNAMIC TYPING

In Python, the type of a variable is determined at runtime, not at compile time.

### Key Points

- You don't need to declare the type of a variable.
- A variable can hold different types of values at different times.
- Python automatically manages memory and type changes during execution.

### Examples

```
x = 10              # int
print(type(x))      # <class 'int'>

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

x = 3.14            # float
print(type(x))      # <class 'float'>

x = [1, 2, 3]       # list
print(type(x))      # <class 'list'>
```

### Another Example

```
age = 25
name = "Alice"
age = "twenty five"     # same variable, new type
is_active = True
is_active = 0           # now it becomes int
```

## 2. STRONGLY TYPED

Python does not allow operations between incompatible types.

### Key Points

- Every value has a specific type.
- Python prevents mixing incompatible types in operations.
- Helps in catching errors and prevents unintended behavior.

### Examples

**Valid Operations (Same or Compatible Types)**

```
a = 10
b = 20
print(a + b)          # 30

c = 3.5
d = 2.5
print(c + d)          # 6.0

s1 = "Hello"
s2 = "Python"
print(s1 + " " + s2)  # Hello Python
```

**Invalid Operations (Different Types)**

```
a = 10
b = "20"
print(a + b)
# TypeError: unsupported
# operand type(s) for +:
# 'int' and 'str'

x = 5
y = [1, 2, 3]
print(x + y)
# TypeError: unsupported
# operand type(s) for +:
# 'int' and 'list'
```

### Why is this combination powerful?

- Dynamic Typing → More flexibility, less code.
- Strong Typing → Fewer bugs, safer programs.
- Ideal balance of programmer productivity and program reliability.

## Dynamic vs Strongly Typed

| Feature | Dynamic Typing | Strongly Typed |
|---|---|---|
| Type Checking Time | At Runtime | During Operation |
| Variable Type | Can change during program execution | Type is fixed for a value |
| Declaration | No need to declare type | Type is inherent to value |
| Allows Mixing Types | Yes (variable can hold any type) | No (operations on incompatible types are not allowed) |
| Example | `x = 10; x = "Hello"; x = 3.14` | `10 + '20'` ✗, `10 + 20` ✓ |

### Real-World Analogy

- Dynamic Typing is like a box that can hold anything at different times.
- Strong Typing is like rules that allow only compatible items to be combined or used together.

### Common Mistakes

- Thinking Python ignores types completely.
  → It checks types during operations.
- Expecting automatic conversions.
  → Convert explicitly using type conversion functions.
- Mixing types in operations.
  → Leads to TypeError.

### Type Conversion is Explicit

```
a = 10
b = "20"
c = a + int(b)      # Works: 30
d = str(a) + b      # Works: "1020"
```

### Check Types Anytime

```
x = [1, 2, 3]
print(type(x))                  # <class 'list'>
print(isistance(x, list))       # True
print(isistance(x, str))        # False
```

`isistance()` is useful to check type or class.

### Key Takeaways

- Python is Dynamic → types decided at runtime.
- Python is Strongly Typed → no invalid operations between incompatible types.
- Makes Python easy to write and safe to run!

**Dynamic gives you freedom, Strong typing gives you safety.**
**That's what makes Python simple, powerful and reliable!**

**Pro Tip:** When in doubt, check the type!
Use `type()` and `isistance()` to write better Python code.

# Control Flow Statements: The if, if-else


Control Flow Statements allow a program to make **decisions** and execute different blocks of code based on **conditions**.

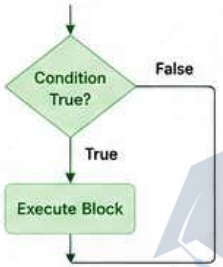
## 1. The if Statement

The if statement is used to execute a block of code only if a condition is True.

### Syntax

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

### Flowchart




### Example 1: Check if a number is positive

```
num = 8
if num > 0:
    print("Number is positive")
```

Output

```
Number is positive
```

### Example 2: Check if a user is eligible to vote

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

Output

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

### Important Notes

- The condition after `if` must be `True` or `False`.
- If the condition is `True`, the indented block runs.
- If `False`, the block is skipped.
- Indentation (usually 4 spaces) is mandatory in Python.

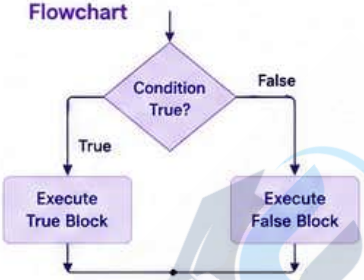
## 2. The if-else Statement

The if-else statement is used to execute one block of code if the condition is True, and another block if it is False.

### Syntax

```
if condition:
    # block of code if True
else:
    # block of code if False
```

### Flowchart




### Example 1: Check if a number is positive or non-positive

```
num = -5
if num > 0:
    print("Positive")
else:
    print("Non-positive")
```

Output

```
Non-positive
```

### Example 2: Find bigger of two numbers

```
a = 12
b = 20
if a > b:
    print("a is bigger")
else:
    print("b is bigger")
```

Output

```
b is bigger
```

## 3. How They Work

- Python evaluates the condition.
- If the condition is **True**, the if block executes.
- If the condition is **False**:
  - In `if`: nothing happens (block skipped).
  - In `if-else`: the else block executes.

### if vs if-else

| Feature | if Statement | if-else Statement |
|---|---|---|
| Purpose | Executes code only if condition is True. | Executes code if True, otherwise executes another block. |
| Else Part | No | Yes |
| Use When | When only one condition needs to be handled. | When two possibilities need to be handled. |
| Example Use | Check if a number is positive. | Check if a number is positive or negative. |

## Common Mistakes

- Forgetting the colon (:) after if or else.
- Wrong indentation (leads to IndentationError).
- Using = instead of == in conditions.
- Expecting else to run when if condition is True.

## More Examples

### Example: Check even or odd

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

### Example: Grade based on marks

```
marks = 85
if marks >= 40:
    print("Pass")
else:
    print("Fail")
```

## Quick Rules

- Use `if` for one-way decisions.
- Use `if-else` for two-way decisions.
- Conditions can use comparison, logical, and membership operators.
- Keep your code clean and properly indented.

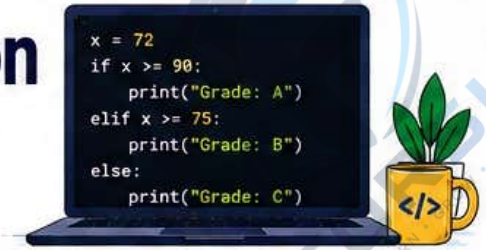
## Key Takeaway

The `if` statement lets your program act when a condition is True.

The `if-else` statement lets your program choose between two paths.

**Decisions make programs smart!**
**Use `if` and `if-else` to control the flow.**

# if-elif-else Decision Control Statement in Python




The `if-elif-else` statement is used to make decisions among multiple conditions.
Python checks the conditions from top to bottom and executes the first True block.

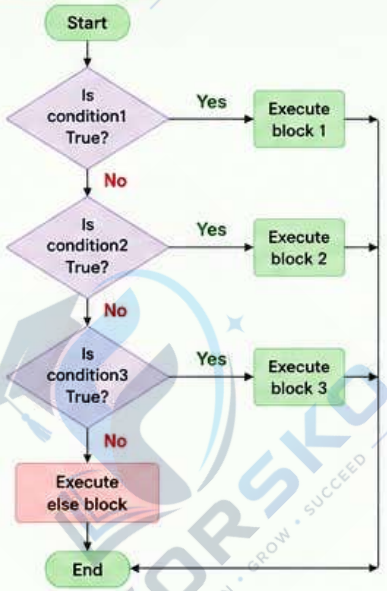
## 1. Syntax

```
if condition1:
    # block of code if condition1 is True
elif condition2:
    # block of code if condition2 is True
elif condition3:
    # block of code if condition3 is True
    ......
else:
    # block of code if all conditions are False
```

### Key Points

- The conditions are checked in order.
- As soon as one condition is True, its block executes and the rest are skipped.
- else is optional, but it is executed if none of the conditions are True.
- Indentation is mandatory.

## 2. Flowchart




## 3. Example Program: Grade Finder

```
marks = int(input("Enter your marks: "))
if marks >= 90:
    print("Grade: A (Excellent)")
elif marks >= 75:
    print("Grade: B (Good)")
elif marks >= 50:
    print("Grade: C (Average)")
else:
    print("Grade: F (Fail)")
```

### Sample Runs

| Input (marks) | Output |
|---|---|
| 95 | Grade: A (Excellent) |
| 80 | Grade: B (Good) |
| 60 | Grade: C (Average) |
| 30 | Grade: F (Fail) |

## 4. How it Works

1. Python checks the first condition (marks >= 90).
2. If True → executes that block and stops.
3. If False → moves to the next condition (marks >= 75).
4. This continues until one condition is True.
5. If none are True → else block executes.

## 5. Truth Conditions (Example)

| marks >= 90 | marks >= 75 | marks >= 50 | Executed Block |
|---|---|---|---|
| True | – | – | if block |
| False | True | – | elif (>=75) block |
| False | False | True | elif (>=50) block |
| False | False | False | else block |

## 6. More Examples

### A. Find Largest of 3 Numbers

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))

if a >= b and a >= c:
    print("a is largest")
elif b >= c:
    print("b is largest")
else:
    print("c is largest")
```

### B. Day Name from Number

```
day = int(input("Enter day (1-7): "))
if day == 1:
    print("Monday")
elif day == 2:
    print("Tuesday")
elif day == 3:
    print("Wednesday")
elif day == 4:
    print("Thursday")
elif day == 5:
    print("Friday")
elif day == 6:
    print("Saturday")
elif day == 7:
    print("Sunday")
else:
    print("Invalid day number")
```

### C. Discount Calculator

```
amount = float(input("Enter amount: "))
if amount >= 5000:
    discount = 0.20
elif amount >= 2000:
    discount = 0.10
elif amount >= 1000:
    discount = 0.05
else:
    discount = 0.0

payable = amount - (amount * discount)
print("Payable amount:", payable)
```

## 7. Common Mistakes

- Forgetting colons (:) after if, elif, else.
- Wrong indentation → IndentationError.
- Using = instead of == for comparison.
- Placing else before elif.
- Expecting multiple conditions to run. (Only the first True block runs.)

### Wrong Example

```
if x > 10
    print("x is greater")
elif x > 5
    print("x is greater than 5")
else
    print("Small number")
```

- Missing colons
- Wrong indentation

## 8. Quick Takeaway

- Use if for single condition.
- Use if-elif-else for multiple conditions.
- Conditions are checked from top to bottom.
- First True block executes.
- else runs only when all conditions are False.

### Pro Tips

- Keep conditions simple and in logical order.
- Use meaningful condition expressions.
- Use elif instead of multiple if to avoid unnecessary checks.

Think → Check → Decide → Execute

Use `if-elif-else` to make your programs smart and efficient!

Practice more, logic more, bug less!

# Nested if Statement in Python

```
x = 15
if x > 0:
    if x % 2 == 0:
        print("Positive Even")
    else:
        print("Positive Odd")
else:
    print("Non-positive")
```

A nested if statement is an if statement inside another if, elif or else block.
It is used to make more complex decisions by checking multiple conditions.

## 1. What is Nested if?

When a condition is True, Python moves inside the block and checks the next condition.

### Key Points

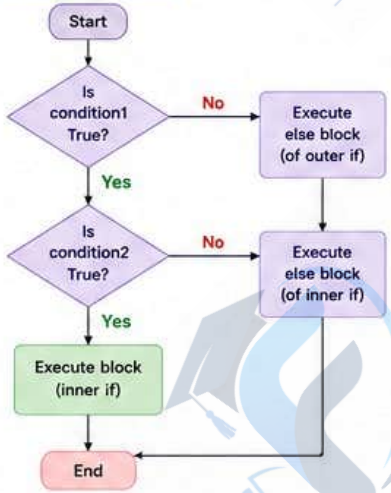
- The inner if is executed only when the outer if (or elif) condition is True.
- It helps to test multiple conditions step by step.
- You can have multiple levels of nesting.
- Proper indentation is very important.

## 2. Syntax and Structure

```
if condition1:
    if condition2:
        # block of code if both condition1 and condition2 are True
    else:
        # block of code if condition2 is False
else:
    # block of code if condition1 is False
```

You can also use elif with nesting.
The inner block can have if, elif, else again.

## 3. Flowchart




## 4. Example Program with Explanation

```
1  num = int(input("Enter a number: "))
2
3  if num > 0:                          # Outer if
4      if num % 2 == 0:                 # Inner if
5          print("Positive Even Number")
6      else:                            # Inner else
7          print("Positive Odd Number")
8  else:                                # Outer else
9      print("Non-positive Number")
```

### Sample Runs

| Input (num) | Output | Explanation |
|---|---|---|
| 10 | Positive Even Number | Outer if True, Inner if True |
| 7 | Positive Odd Number | Outer if True, Inner if False |
| -3 | Non-positive Number | Outer if False |

## 5. More Examples

### A. Check Grade Based on Marks

```
marks = int(input("Enter marks (0-100): "))
if marks >= 60:
    if marks >= 90:
        print("Grade: A")
    elif marks >= 75:
        print("Grade: B")
    else:
        print("Grade: C")
else:
    print("Grade: F (Fail)")
```

First checks if passed (>=60).
If passed, then checks for A or B, otherwise C. If not passed, then F.

### B. Check Eligibility for Driving

```
age = int(input("Enter age: "))
has_license = input("Do you have license? (y/n): ")
if age >= 18:
    if has_license.lower() == 'y':
        print("You can drive.")
    else:
        print("You need a license.")
else:
    print("You must be 18 or older.")
```

Outer if checks age.
If age condition True, inner if checks license.

### C. Largest of Three Numbers

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))
if a >= b:
    if a >= c:
        print("Largest is", a)
    else:
        print("Largest is", c)
else:
    if b >= c:
        print("Largest is", b)
    else:
        print("Largest is", c)
```

Compare a and b first, then compare with c.

## 6. Common Mistakes

- Incorrect indentation
  Leads to IndentationError.
- Forgetting colon (:) after if or else.
- Placing else without matching if.
  Else must belong to the naerest if.
- Over-complicating logic.
  Keep conditions simple and readable.

### Wrong Example

```
if x > 0                    ⟷ Missing colon
    if x % 2 == 0:
        print("Even")
    else:                   ⟷ Wrong indentation
        print("Odd")
```

### Correct Example

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

## 7. When to Use Nested if

- When a condition depends on another condition being True.
- When decisions have multiple levels.
- When you need more precise control over the flow of execution.

### Quick Takeaway

- Nested if = decision within a decision.
- Check conditions from outer to inner.
- Only True paths go deeper.
- Good indentation = bug free code!

Break complex problems into smaller decisions using Nested if.

Think step by step. Code smart!

# The while Loop in Python

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

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

The while loop is used to execute a block of code repeatedly as long as the specified condition is True.

## 1. What is a while Loop?

The while loop repeats a block of code as long as the condition is True. When the condition becomes False, the loop stops.

### Key Points

- Condition is checked before each iteration.
- If condition is False initially, loop body will not execute.
- Used when the number of iterations is unknown.
- Proper update of the condition is essential to avoid infinite loops.

## 2. Syntax

```
while condition:
    # block of code
    # update condition
```

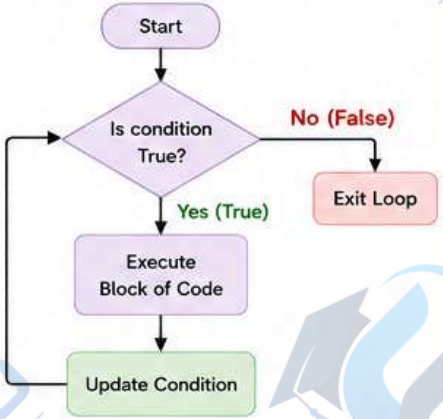
### Example Syntax

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

### Equivalent to:

Repeat the block of code while condition remains True.

## 3. How it Works?




## 4. Example: Print 1 to 5

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

### Output

```
1
2
3
4
5
```

### Step-by-step Execution

| Iteration | count (before) | Condition (count <= 5) | Action | count (after) |
|---|---|---|---|---|
| 1 | 1 | True | print 1 | 2 |
| 2 | 2 | True | print 2 | 3 |
| 3 | 3 | True | print 3 | 4 |
| 4 | 4 | True | print 4 | 5 |
| 5 | 5 | True | print 5 | 6 |
| Next | 6 | False | Exit loop | -- |

## 5. More Examples

### A. Sum of first n natural numbers

```
n = int(input("Enter n: "))
i = 1
sum = 0
while i <= n:
    sum += i
    i += 1
print("Sum =", sum)
```

Example:
Enter n: 5
Sum = 15

### B. Factorial of a number

```
n = int(input("Enter n: "))
fact = 1
i = 1
while i <= n:
    fact *= i
    i += 1
print("Factorial =", fact)
```

Example:
Enter n: 5
Factorial = 120

### C. Count digits in a number

```
num = int(input("Enter a number: "))
count = 0
while num > 0:
    num //= 10
    count += 1
print("Number of digits =", count)
```

Example:
Enter a number: 12345
Number of digits = 5

## 6. Common Mistakes

- **Infinite Loop:** Forgetting to update the condition inside the loop.

```
i = 1
while i <= 5:
    print(i)      # i is never updated
```

- **Wrong Condition:** Using wrong operators.

```
while i = 5:   # = is assignment, not comparison
```

- **Indentation Error:** Incorrect indentation of loop body.

```
while i <= 5:
print(i)
```

```
while i <= 5:
    print(i)
```

## 7. Useful Variations

- `while True:`
  `# Infinite loop (use break to stop)`
- Using `break`:

```
i = 1
while i <= 10:
    if i == 7:
        break        # exits loop when i == 7
    print(i)
    i += 1
```

- Using `continue`:

```
i = 1
while i <= 5:
    i += 1
    if i == 3:
        continue     # skip printing 3
    print(i)
```

## 8. When to Use while Loop?

- When the number of repetitions is not known in advance.
- When we need to keep repeating until a condition changes.
- When reading input until a specific condition is met.

## Key Takeaway

while loop keeps running as long as the condition is True.

Update the condition → Avoid infinite loop → Write correct logic.

## Pro Tip

Always test your loop with small inputs first and make sure your condition will eventually become False.

# The for Loop in Python

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

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

The for loop is used to iterate (repeat) over a sequence (such as list, tuple, string, range, etc.) or other iterable objects.

## 1. What is a for Loop?

The for loop is used to execute a block of code for each item in a sequence (or for a fixed number of times using `range()`).

### Key Points

- It is used when the number of iterations is known.
- It automatically moves to the next item after each iteration.
- No need to update a counter manually.
- Cleaner and easier to read than while loops in many cases.

## 2. Syntax

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

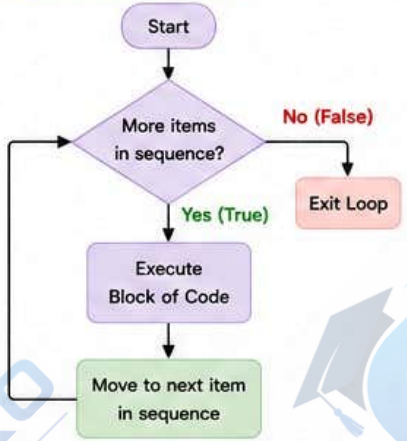
### Explanation

- **'variable'** takes the value of each item one by one.
- **'sequence'** can be any iterable object.

### Example Syntax

```
for x in [10, 20, 30]:
    print(x)
```

## 3. How it Works?




## 4. Example Program: Print items of a list

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

**Output**

```
apple
banana
mango
```

### Step-by-step Execution

| Iteration | fruit (current item) | Action | Output so far |
|---|---|---|---|
| 1 | apple | print(apple) | apple |
| 2 | banana | print(banana) | apple<br>banana |
| 3 | mango | print(mango) | apple<br>banana<br>mango |
| Next | – | No more items → Exit loop | – |

## 5. More Examples

### A. Using range()

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

Output:

```
1
2
3
4
5
```

### B. Sum of first n natural numbers

```
n = int(input("Enter n: "))
s = 0
for i in range(1, n+1):
    s += i
print("Sum =", s)
```

Example (n = 5)
Output: Sum = 15

### C. Iterate over a string

```
for ch in "Python":
    print(ch, end=' ')
```

Output:

```
P y t h o n
```

### D. Iterate over a tuple

```
t = (10, 20, 30)
for x in t:
    print(x)
```

Output:

```
10
20
30
```

## 6. Common Mistakes

- Forgetting the colon (:) after for statement.

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

- Wrong indentation.

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

- Using range() incorrectly.

```
for i in range(5, 1):  # won't run (step is positive)
    print(i)
```

- Modifying the loop variable inside the loop.

```
for i in range(5):
    i = i + 1    # has no effect on next iteration
```

## 7. Useful Variations

### Using range(start, stop, step)

```
for i in range(2, 11, 2):   # 2 to 10, step 2
    print(i)
```

Output: 2 4 6 8 10

### Using enumerate() (index + value)

```
names = ["Ali", "Sara", "John"]
for idx, name in enumerate(names):
    print(idx, name)
```

Output:

```
0 Ali
1 Sara
2 John
```

### Loop with else

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

Output:

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

## 8. When to Use for Loop?

- When you know how many times to repeat.
- When you are iterating over a sequence (list, tuple, string, set, dict, etc.).
- When you need cleaner and more readable code.

### Key Takeaway

The for loop makes your code short, clean and efficient when the number of iterations is known.
Use it to iterate, process and automate tasks easily!

### Pro Tip

- Use range() for numbers.
- Use for with collections to access each item.
- Prefer for over while when the count is known.

# The continue and break Statements in Python

```
for i in range(1, 6):
    if i == 3:
        continue  # skip 3
    if i == 4:
        break     # stop at 4
    print(i)
# Output: 1 2
```

Both continue and break are used to control the flow of loops (for and while).
They help us skip iterations or terminate the loop before it completes.

## 1. continue Statement

The continue statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

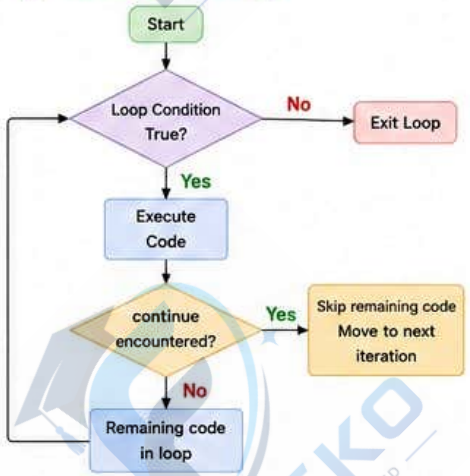
### Syntax

```
continue
# used inside a loop
```

### How it works?

- Rest of the code in the loop is skipped.
- Control moves to the next iteration.
- Loop does not terminate.

### Flowchart (continue)




### Examples with continue

**A. Skip even numbers and print odd numbers**

```
for i in range(1, 8):
    if i % 2 == 0:
        continue    # skip even numbers
    print(i)
```

Output:
```
1
3
5
7
```

Here, when i is even, continue skips print(i) and moves to the next number.

**B. Skip a particular value**

```
for name in ["Ali", "Sara", "John", "Eva"]:
    if name == "Sara":
        continue    # skip Sara
    print(name)
```

Output:
```
Ali
John
Eva
```

When name is "Sara", it is skipped and loop continue with the next item.

## 2. break Statement

The break statement terminates the loop completely and transfers the control to the statement immediately after the loop.

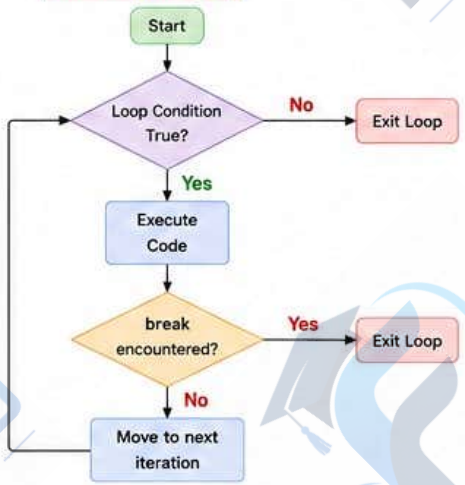
### Syntax

```
break
# used inside a loop
```

### How it works?

- When break is encountered, the loop stops.
- No more iterations are executed.
- Control moves to the next statement after the loop.

### Flowchart (break)




### Examples with break

**A. Stop the loop when a condition is met**

```
for i in range(1, 10):
    if i == 5:
        break    # stop when i is 5
    print(i)
```

Output:
```
1
2
3
4
```

When i becomes 5, break is executed and loop terminates.

**B. Search and stop when item is found**

```
numbers = [4, 7, 2, 9, 5]
for x in numbers:
    if x == 9:
        print("Found 9")
        break
    print(x)
```

Output:
```
4
7
2
Found 9
```

When 9 is found, break stops the loop.

## 3. continue vs break

| Feature | continue | break |
|---|---|---|
| Purpose | Skips current iteration | Terminates the loop |
| Effect | Moves to next iteration | Exits the loop completely |
| Loop | Loop continues | Loop stops |
| Use When | You want to skip some iterations | You want to stop the loop based on a condition |

## 4. Common Mistakes

- Forgetting indentation after continue or break.
- Using continue or break outside a loop (leads to SyntaxError).
- Expecting break to skip only current iteration (it stops the loop).
- Forgetting that code after continue will not run in that iteration.

**Wrong Example**

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

print(i) is never executed. (It is after continue.)

**Correct Example**

```
for i in range(5):
    continue
    # next iteration starts
```

## 5. Key Takeaways

- Use continue to skip unwanted iterations.
- Use break to terminate the loop early.
- Both are very useful for writing efficient and clean code.
- They work with both for and while loops.

### Pro Tip

Use continue to avoid deep nesting of if conditions.
Use break when the goal is achieved and no more searching is needed.

Remember: Loops are powerful, but controlling them correctly with continue and break makes your code smarter and more effective!


1010101010
0101010101
1010101010
0101010101
Unit - III
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

# Functions: Built-In Functions in Python

```
print(len("Python"))      # 6
print(max([4, 1, 7, 2]))  # 7
print(type(10.5))         # <class 'float'>
```

Built-in functions are pre-defined in Python and are always available for use.
They perform common tasks, making our programs shorter, easier and more efficient.

## 1. What are Built-In Functions?

These functions are part of Python's standard library. You can use them directly without importing any module.

**Example:** `print()`, `len()`, `type()`, `max()`, `min()`, `sum()`, `input()` etc.

## 2. Categories of Built-In Functions

- Input / Output Functions
- Type Conversion Functions
- Numeric Functions
- Sequence (String / List / Tuple) Functions
- Set / Dictionary Functions
- Utility Functions
- Others

## 4. Commonly Used Built-In Functions with Examples

| Category | Function | Description | Example | Output |
|---|---|---|---|---|
| Input / Output Functions | print() | Displays the given value(s) on the screen. | print("Hello Python") | Hello Python |
| | input() | Takes input from the user as a string. | name = input("Enter name: ") | (user input) |
| Type Conversion Functions | int() | Converts value to integer. | int("25") | 25 |
| | float() | Converts value to float. | float("3.14") | 3.14 |
| | str() | Converts value to string. | str(100) | '100' |
| | type() | Returns the type of an object. | type(10.5) | <class 'float'> |
| Numrce Functions | abs() | Returns absolute value. | abs(-9) | 9 |
| | pow(x, y) | Returns x raised to the power y. | pow(2, 3) | 8 |
| | round(x, n) | Rounds x to n digits. | round(3.14159, 2) | 3.14 |
| | max(iterable) | Returns largest item. | max([4, 1, 7, 2]) | 7 |
| | min(iterable) | Returns smallest item. | min("python") | 'h' |
| | sum(iterable) | Returns sum of items. | sum([1, 2, 3, 4]) | 10 |
| Sequence (String / List / Tuple) Functions | len(s) | Returns length (number of items). | len([10, 20, 30]) | 3 |
| | sorted(iterable) | Returns a sorted list. | sorted([3, 1, 2]) | [1, 2, 3] |
| | reversed(seq) | Returns reverse iterator. | list(reversed("abc")) | ['c', 'b', 'a'] |
| | enumerate(seq) | Returns index and value as pairs. | list(enumerate(['a','b"])) | [(0, 'a'), (1, 'b')] |
| Set / Dictionary Functions | len(set/dict) | Returns number of items. | len({'a':1, 'b':2}) | 2 |
| | keys() | Returns all keys of dict. | {'a':1,'b':2}.keys() | dict_keys(['a', 'b']) |
| | values() | Returns all values of dict. | {'a':1,'b':2}.values() | dict_values([1, 2]) |
| | items() | Returns key-value pairs. | {'a':1}.items() | dict_items([('a', 1)]) |
| Utility Functions | all(iterable) | Returns True if all items are True. | all([True, 1, 'yes"]) | True |
| | any(iterable) | Returns True if any item is True. | any([0, False, 3]) | True |
| Others | id(obj) | Returns identity of object (unique address). | id(10) | (some integer) |
| | help(obj) | Shows help/documentation for an object. | help(len) | (help text) |
| | dir(obj) | Returns list of attributes and methods of object. | dir([]) | (list of attributes) |

## 5. Examples in Action

```
# Length and type
text = "Python"
print(len(text))
print(type(text))
```

Output:
```
6
<class 'str'>
```

```
# Numeric functions
nums = [5, 9, 2, 8]
print(max(nums))
print(min(nums))
print(sum(nums))
```

Output:
```
9
2
24
```

```
# Type conversion
a = int("42")
b = float("3.5")
c = str(100)
print(a, b, c)
```

Output:
```
42 3.5 100
```

```
# Sequence functions
lst = [10, 20, 30]
print(sorted(lst))
print(reversed(lst))
```

Output:
```
[10, 20, 30]
<list_reverseiterator
  object at 0x...>
```

```
# Dictionary functions
d = {'x': 1, 'y': 2}
print(d.keys())
print(d.values())
```

Output:
```
dict_keys(['x', 'y'])
dict_values([1, 2])
```

## 6. Common Mistakes

- Forgetting parentheses: `print` instead of `print()`
- Using wrong type in function: `max(10)` (max needs iterable)
- Expecting original list to change with `sorted()` (It returns a new sorted list)
- Confusing type conversion: `int("3.5")` ✗ (use `float` for decimals)

## 7. Key Takeaways

- Built-in functions save time and effort.
- They are easy to use and very powerful.
- Know the right function for the right task!

## . Notes

- Function names are case-sensitive.
- Some functions work on specific data types.
- Use `help(function_name)` for details. Example: `help(len)`
- All built-in functions can be used without importing any module.

**Pro Tip:** Use `help(function_name)` to learn more about any built-in function. **Example:** `help(sum)`

"Smart use of built-in functions makes your code Pythonic and efficient!"

# Commonly Used Modules in Python

Modules are Python files containing functions and variables.
They extend Python's functionality and help us write clean, reusable and efficient code.

```python
# Using a module in Python
import math
print(math.sqrt(16))   # 4.0
print(math.pi)         # 3.141592653589793
```

## 1. What is a Module?

- A module is a Python file with `.py` extension that contains definitions (functions, classes, variables, etc.).
- We can import a module to use its functionality.
- **Example:**

```python
import math       # import the math module
print(math.sqrt(25))    # 5.0
```

## 2. How to Use Modules?

1. Using the entire module
```python
import module_name
```
2. Importing specific functions/objects
```python
from module import name1, name2
```
3. Importing with an alias
```python
import module_name as alias
```
4. Importing everything (not recommended)
```python
from module import *
```

## 3. List of Commonly Used Modules

| Module | Purpose |
|---|---|
| math | Mathematical functions |
| random | Generate random numbers |
| datetime | Work with dates and times |
| time | Time related functions |
| os | Operating system interface |
| sys | System-specific parameters |
| re | Regular expressions (pattern matching) |
| json | Work with JSON data |
| collections | Specialized container datatypes |
| statistics | Statistical operations |

## 4. Top Modules with Examples

| Module | What it Does | Commonly Used Functions / Attributes | Example | Output |
|---|---|---|---|---|
| math | Provides mathematical functions and constants. | `sqrt()`, `pow()`, `factorial()`, `sin()`, `cos()`, `log()`, `pi`, `e` | `import math`<br>`print(math.sqrt(16))`<br>`print(math.pi)` | `4.0`<br>`3.141592653589793` |
| random | Generates random numbers and performs random operations. | `random()`, `randint()`, `choice()`, `shuffle()` | `import random`<br>`print(random.randint(1, 10))`<br>`print(random.choice(['a','b','c']))` | `7` (varies)<br>`'b'` |
| datetime | Manipulates dates and times. | `datetime.now()`, `date.today()`, `timedelta()` | `from datetime import datetime`<br>`print(datetime.now())` | `2024-06-01 12:30:45.123456` (varies) |
| time | Provides time related functions. | `time()`, `sleep()`, `ctime()` | `import time`<br>`print(time.time())` | `1717234567.85` (epoch time) |
| os | Interacts with the operating system. | `listdir()`, `mkdir()`, `remove()`, `path.exists()` | `import os`<br>`print(os.listdir())` | `['file1.py', 'notes.txt', 'data']` (varies) |
| sys | Provides access to system specific parameters. | `argv`, `version`, `exit()`, `path` | `import sys`<br>`print(sys.version)` | `3.12.2 (main, Feb 6 2024, 10:00:00) [GCC ...]` |
| re | Supports regular expression operations. | `search()`, `match()`, `findall()`, `sub()` | `import re`<br>`text = "My number is 123"`<br>`print(re.findall(r'\d+', text))` | `['123']` |
| json | Encodes and decodes JSON data. | `dumps()`, `loads()` | `import json`<br>`data = {'name': 'Aman', 'age': 20}`<br>`print(json.dumps(data))` | `{"name": "Aman", "age": 20}` |
| collections | Provides specialized container datatypes. | `Counter`, `defaultdict`, `namedtuple`, `deque` | `from collections import Counter`<br>`c = Counter('banana')`<br>`print(c)` | `Counter({'a': 3, 'n': 2, 'b': 1})` |
| statistics | Performs statistical calculations. | `mean()`, `median()`, `mode()`, `stdev()` | `import statistics`<br>`nums = [2, 4, 4, 4, 5, 5, 7]`<br>`print(statistics.mean(nums))` | `4.428571428571429` |

## 5. Example Programs

**Program: Roll a dice (random module)**

```python
import random
print(random.randint(1, 6))
```

Output:
```
4
(varies)
```

**Program: Current date and time (datetime module)**

```python
from datetime import datetime
now = datetime.now()
print(now.strftime("%d-%m-%Y %H:%M:%S"))
```

Output:
```
01-06-2024
12:45:30
(varies)
```

**Program: List files in current directory (os module)**

```python
import os
files = os.listdir('.')
print(files)
```

Output:
```
['a.py', 'b.txt',
'images', ...]
(varies)
```

**Program: Find all email IDs in a text (re module)**

```python
import re
text = "Contact us at test@gmail.com
   or info@site.com"
emails = re.findall(r'[\w.]+@[\w.]+', text)
print(emails)
```

Output:
```
['test@gmail.com',
'info@site.com']
```

## 6. Common Mistakes

- Forgetting to import the module. — `print(math.sqrt(4))  # Error: name 'math' is not defined`
- Using * import (pollutes namespace). — `from math import *  # Not recommended`
- Wrong module name (case sensitive). — `import Math  # Error (should be 'math')`
- Expecting a module to be available without installing (external modules). — Install using `pip install module_name`

## 7. Key Takeaways

- Modules extend Python's power and reduce our effort.
- Import only what you need.
- Use the right module for the right task.
- Built-in modules are ready to use (no installation needed).
- Explore `help(module_name)` to learn more.

## 8. Helpful Tips

- Use `dir(module_name)` to see all functions and attributes.
- Use `help(module_name.function)` for detailed help.
- Read official docs: https://docs.python.org/3/library/

**Modules are Python's superpowers! Learn, use and explore them to become a better programmer.**

# Function Definition and Calling the Function

```python
def greet(name):
    print(f"Hello, {name}!")

greet("Python") # Function call
greet("Developer")
```

A function is a block of organized, reusable code that performs a specific task.
We first define (create) the function and then call (use) it whenever needed.

## 1. What is a Function?

- A function is a named block of code that performs a specific task.
- It improves code reusability, readability and reduces repetition.
- Functions are executed only when they are called.

## 2. Why Use Functions?

- Avoid repeating the same code.
- Make programs easier to read and maintain.
- Break large programs into smaller parts.
- Promotes code reusability.

## 3. Anatomy of a Function

```
def function_name(parameters):      ← Header
    """Docstring (optional)"""      ← Docstring
    # statements (function body)    ← Body
    return value  # (optional)      ← Return
```

## 4. Function Definition (Syntax)

```
def function_name(parameters):
    """Docstring (optional)"""
    # statements
    return value  # (optional)
```

**Explanation**

- **def** – keyword to define a function
- **function_name** – any valid name
- **parameters** – inputs to the function (optional)
- **return** – sends a value back (optional)

## 5. Calling the Function

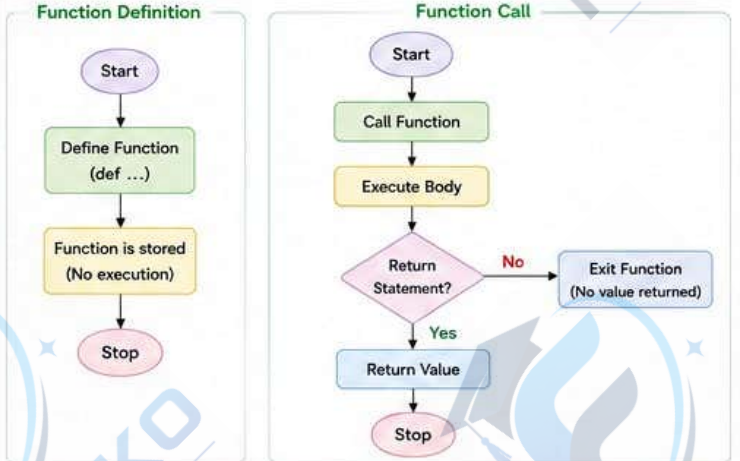
```
result = function_name(arguments)
```

- **function_name** – name of the function
- **arguments** – actual values passed to the function
- **result** – (optional) variable to store returned value

**Explanation**

- When a function is called, its body executes.
- If it has a **return** statement, the value is returned to the caller.
- If **return** is not used, the function performs tasks without returning value.

## 6. Flow of Execution




## 7. Common Mistakes

- Forgetting the colon (:) after def.
- Indentation errors.
- Calling the function before it is defined.
- Mismatching number of arguments.
- Forgetting return when a value is expected.

```
# Error Example
def add(a, b)    # Missing colon
    return a + b
```

## 8. Examples

**1. Function with No Parameters**

```
def say_hello():
    print("Hello, World!")

Call:
say_hello()

Output:
Hello, World!
```

**2. Function with Parameters**

```
def greet(name):
    print("Hello, " + name + "!")

Call:
greet("Alice")

Output:
Hello, Alice!
```

**3. Function with Multiple Parameters**

```
def add(a, b):
    return a + b

Call:
result = add(5, 7)
print(result)

Output:
12
```

**4. Function with Return Value**

```
def square(n):
    return n * n

Call:
s = square(6)
print(s)

Output:
36
```

**5. Function Without Return**

```
def show(name):
    print("Welcome,", name)

Call:
show("Bob")

Output:
Welcome, Bob
```

## 9. More Examples

**Even or Odd Checker**

```
def check(n):
    if n % 2 == 0:
        return "Even"
    else:
        return "Odd"

print(check(10))  # Even
print(check(7))   # Odd
```

**Factorial Function**

```
def fact(n):
    f = 1
    for i in range(1, n+1):
        f = f * i
    return f

print(fact(5))  # 120
```

## 10. Things to Remember

- Define a function once, use it many times.
- Use meaningful names for functions and parameters.
- Use return when you need to get a value back from the function.
- Functions can call other functions.
- Keep functions short and focused on one task.

## 11. Key Takeaways

- A function is defined using the def keyword.
- It can take parameters and return a value.
- You call a function by its name followed by parentheses.
- Functions make code modular, reusable and easy to manage.
- Practice creating and using functions to build better programs!

**Good functions make your code DRY (Don't Repeat Yourself) and your programs SMART!**

# The return Statement and void Function in Python

Functions can either return a value to the caller using return statement, or perform an action without returning anything (void function).

```
def add(a, b):
    return a + b

def greet(name):
    print("Hello,", name)
    # No return -> void function
```

## 1. What is return Statement?

- The return statement sends a value from the function back to the caller.
- It can return any data type.
- When return is executed, the function terminates immediately.

## 2. What is void Function?

- A void function does not return any value to the caller.
- It performs a task (e.g., print, update, calculate) and exits.
- In Python, if a function has no return statement, it returns None by default.

## 3. The return Statement

### Syntax

```
def function_name(parameters):
    # statements
    return value   # optional
```

### Examples

**1. Return a value**

```
def square(n):
    return n * n

result = square(5)
print(result)
```

Output: 25

**2. Return multiple types**

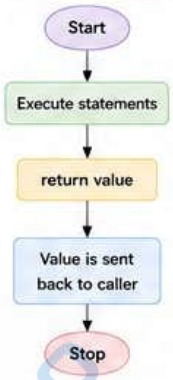
```
def get_info():
    return "Python", 3.11, Tutput:

lang, ver, status = get_info()
print(lang, ver, status)
```

**3. Early return**

```
def check(n):
    if n < 0:
        return "Negative"
    return "Non-negative"

print(check(-10))
print(check(7))
```

Output:
# Non-negative

### Flow of return




## 4. void Function (No return Value)

### Syntax

```
def function_name(parameters):
    # statements
    # No return statement
```

### Examples

**1. Print a message**

```
def greet(name):
    print("Hello,", name)

greet("Alice")
print(greet("Bob"))
```

Output:
Hello, Alice
Hello, Bob
None

**2. Perform a task (no return)**

```
def add_and_print(a, b):
    s = a + b
    print("Sum =", s)

add_and_print(4, 6)
```

Output: Sum = 10

**3. Default return value is None**

```
def do_nothing():
    x = 10
    y = 20
    # no return

res = do_nothing()
print(res)
```

Output: None

### Flow of void function


## 5. When to Use?

**Use return when:**

- You need to send a result back to the caller.
- You want to use that result for further calculation.
- Example: mathematical functions, getters, validations.

**Use void function when:**

- You only want to perform an action.
- You don't need to send any value back.
- Example: display messages, logging, updates.

## 6. return vs void Function

| Feature | return Function | void Function |
|---|---|---|
| Purpose | Returns a value to the caller | Does not return any value |
| Syntax | Uses return statement | Has no return statement |
| Returned Value | Any data type | None (by default) |
| Use Case | When result is needed | When only an action is needed |
| Example | `def add(a, b): return a + b` | `def show(): print("Hi")` |
| Caller | Can store or use the returned value | Cannot store anything useful (it returns None) |

## 7. Common Mistakes

**Forgetting return (returns None)**

```
def add(a, b):
    c = a + b   # forgot return

res = add(3, 4)
print(res)   # None
```

**Using return outside function**

```
return 10      # SyntaxError
```

**Expecting value from void function**

```
def print_msg():
    print("Hi")

x = print_msg()
print(x)     # None
```

**Unreachable code after return**

```
def f():
    return 5
    print("This will not run")
```

## 8. Key Takeaways

- Use return to send a value back from a function.
- If a function has no return, it is a void function.
- Void function returns None by default.
- return stops the function immediately.
- Choose return or void based on the task requirement.
- Good functions are clear, focused and do one thing well.

**Pro Tip:** In Python, every function returns something.

With return → useful value → Without return → None

"Great programs are built with great functions!"

# Scope and Lifetime of Variables in Python

**Scope** determines where a variable can be accessed in a program.
**Lifetime** determines how long a variable exists in memory.

```
x = 10          # global scope
def func():
    y = 20      # local to func
    print(x)    # 10  (uses global x)
func()
print(x)        # 10
# print(y)  # Error! y not accessible here
```

## 1. What is Scope?

- The region of a program where a variable is recognized and can be used.
- Python has four types of scope (**LEGB Rule**).

## 2. Types of Scope in Python (LEGB Rule)

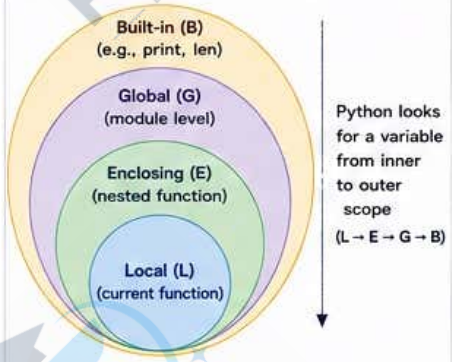
**L** — Local (function scope)
Inside a function. Accessible only within that function.

**E** — Enclosing (non-local scope)
Inside nested functions. Refers to the scope of the enclosing function.

**G** — Global (module scope)
Defined at the top level of a module. Accessible throughout the module.

**B** — Built-in (Python built-in scope)
Names in the built-in namespace (e.g., len, print, int).

## 3. Scope Visualization (LEGB)




## 4. Scope Summary

| Scope | Where Declared | Accessible From | Lifetime | Example Variable |
|---|---|---|---|---|
| Local | Inside a function | Only within that function | Created when the function is called and destroyed when the function ends | `x` inside a function |
| Enclosing (non-local) | Inside an outer (enclosing) function | Inner nested functions | Exists as long as the enclosing function exists | `y` in `outer()` accessed by `inner()` |
| Global | At the top level of a module | Throughout the module (after declaration) | From the time the program starts until it ends | `z` at module level |
| Built-in | Built-in namespace (by Python) | Throughout the program | Exists as long as Python is running | `print`, `len`, `int` |

## 5. Examples

### 5.1 Local Scope

```
def greet():
    msg = "Hello"
    print(msg)

greet()
# print(msg)    # Error
```

**Output:**
```
Hello
```

**Note:** msg exists only inside greet().

### 5.2 Enclosing (Non-local) Scope

```
def outer():
    x = 10      # Enclosing scope
    def inner():
        print(x)  # uses x from outer
    inner()
outer()
```

**Output:**
```
10
```

**Note:** inner() can access x from enclosing function outer().

### 5.3 Global Scope

```
x = 5  # Global variable

def show():
    print(x)  # uses global x

show()
print(x)
```

**Output:**
```
5
5
```

**Note:** x is accessible everywhere in the module.

### 5.4 Built-in Scope

```
print(len("Python"))  # len is built-in
print(max([3, 7, 1]))  # max is built-in
```

**Output:**
```
6
7
```

**Note:** Built-in names are always available.

## 6. Lifetime of Variables

- **Local variables:** Exist only during the execution of the function.
- **Enclosing variables:** Exist as long as the enclosing function is active.
- **Global variables:** Exist from program start until program termination.
- **Built-in variables:** Exist as long as the Python interpreter is running.

### Example (Lifetime)

```
def demo():
    a = 100   # exists during function call
    print(a)

demo()
# print(a)  # Error: a no longer exists
```

**Output:**
```
100
```

## 7. Modifying Variables from Outer Scope

### 7.1 Using global

```
x = 0

def change():
    global x    # declare x as global
    x = 20      # modify global x

change()
print(x)   # 20
```

**Note:** global allows modification of global variable inside a function.

### 7.2 Using nonlocal

```
def outer():
    x = 10

    def inner():
        nonlocal x   # refer to enclosing x
        x = 30

    inner()
    print(x)

outer()   # 30
```

**Note:** nonlocal allows modification of enclosing variable in nested function.

## 8. Order of Resolution (LEGB Rule)

Python searches for a variable in this order:

**Local (L)** → **Enclosing (E)** → **Global (G)** → **Built-in (B)**

The first match is used.

**Example:**

```
x = "global"

def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print(x)      # local
        print(x)      # enclosing
    inner()
outer()
print(x)              # global
```

**Output:**
```
local
enclosing
global
```

## 9. Common Mistakes

- Trying to access a local variable outside the function.
- Forgetting global when modifying global variables.
- Forgetting nonlocal when modifying enclosing variables.
- Assuming variables defined in one function are available in another.

## 10. Key Takeaways

- Scope defines where a variable can be used.
- Lifetime defines how long a variable exists.
- Follow LEGB rule to understand variable resolution.
- Use global or nonlocal to modify outer variables.
- Good understanding prevents bugs and makes code clean and predictable.

**Pro Tip:** Use the smallest possible scope for your variables. It keeps code clean, safe and easier to understand!

**Write better. Think in scopes!**

# Default Parameters in Python

```
def greet(name, msg="Hello"):
    print(f"{msg}, {name}!")

greet("Alice")              # Hello, Alice!
greet("Bob", "Hi")          # Hi, Bob!
```

Default parameters allow you to define a default value for a parameter in a function.
If no argument is passed for that parameter during the function call, the default value is used.

## 1 What are Default Parameters?

- Parameters that have default values in the function definition.
- Make functions more flexible and easy to use.
- If an argument is not provided, Python uses the default value.

## 2 Syntax

```
def function_name(param1, param2=default,
    param3=default, ...):
    # function body
```

## 3 Rules to Remember

1. Default parameters must be at the end of the parameter list.
2. A parameter with a default value is optional during function call.
3. If you provide a value, the default value is ignored.
4. Default values are evaluated only once, when the function is defined.
5. Use immutable values (int, str, float, tuple, bool) as defaults. Avoid mutable types (list, dict, set) as defaults.

## Invalid Example (Rule 1)

Non-default argument follows default argument

```
def wrong(a=10, b):        # X SyntaxError
    pass
# Correct way:
def correct(a, b=10):      #  ✓ Valid
    pass
```

## 5 Examples

### 5.1 Basic Default Parameter

```
def greet(name, msg="Hello"):
    print(f"{msg}, {name}!")

greet("Alice")
greet("Bob", "Hi")
```

- msg has default value "Hello".
- First call uses default.
- Second call uses provided value "Hi".

Output:
```
Hello, Alice!
Hi, Bob!
```

### 5.2 Multiple Default Parameters

```
def add(a, b=0, c=0):
    return a + b + c

print(add(5))
print(add(5, 10))
print(add(5, 10, 15))
```

- b and c have default value 0.
- You can pass 0, 1 or 2 additional arguments.

Output:
```
5
15
30
```

### 5.3 Default Parameter with Different Data Types

```
def info(name, age=18, city="Unknown"):
    print(f"Name: {name}, Age: {age}, City: {city}")

info("Riya")
info("Rohan", 22)
info("Amit", 25, "Delhi")
```

- age is int, city is str.
- Missing arguments are taken from defaults.

Output:
```
Name: Riya, Age: 18, City: Unknown
Name: Rohan, Age: 22, City: Unknown
Name: Amit, Age: 25, City: Delhi
```

### 5.4 Keyword Arguments with Defaults

```
def display(a, b=10, c=20):
    print(a, b, c)

display(1)
display(1, c=30)
display(c=40, a=5)
display(a=7, b=8, c=9)
```

- You can override defaults using keyword arguments.
- Order does not matter when using keywords.

Output:
```
1 10 20
1 10 30
5 10 40
7 8 9
```

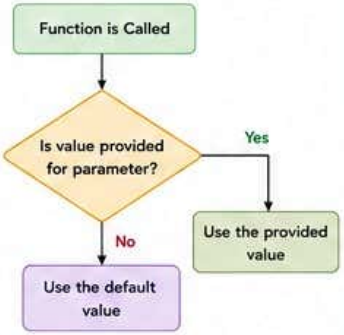
### 5.5 Default Parameter with Expression

```
import math

def circle_area(r, pi=math.pi):
    return pi * r * r

print(circle_area(2))      # uses default pi
print(circle_area(2, 3.14)) # custom pi
```

- Default value can be an expression.
- You can still override it.

Output:
```
12.566370614359172
12.56
```

## 6 How Default Parameters Work? (Execution Flow)




**Key Points**

- Python checks each parameter from left to right.
- If a value is provided, it uses that value.
- If not, it uses the default value defined in the function.
- If no default exists and no argument is provided → Error!

## 7 Default vs Required Parameters

| Feature | Required Parameter | Default Parameter |
|---|---|---|
| Definition | No default value | Has a default value |
| Call Requirement | Must provide value | Optional to provide value |
| Example | `def f(a):` | `def f(a=10):` |
| Flexibility | Less flexible | More flexible |
| Error if missing | Yes | No (default is used) |

## Common Mistakes

- Putting non-default parameter after default parameter.
- Using mutable default arguments (like `list`, `dict`) → leads to bugs!
- Assuming default value changes during function calls.

Bad Example (Mutable Default)

```
def append_item(item, lst=[]):
    lst.append(item)
    return lst

print(append_item(1))  # [1]
print(append_item(2))  # [1, 2] (why?)
```

## 9 Key Takeaways

- Default parameters make functions user-friendly.
- They reduce the need for multiple function versions.
- Place default parameters at the end.
- Use immutable values as defaults.
- They improve code readability and flexibility.

**Pro Tip** Default parameters are evaluated once when the function is defined, not each time the function is called.

### Real-life Analogy

Think of default parameters like a restaurant order form.
If you don't choose a spice level, it serves "Medium" by default!

"Good defaults make great functions!"

# Keyword Arguments in Python

```
>>> def greet(name, msg):
        print(f"Hello {name}, {msg}")

>>> greet(name="Alice", msg="Welcome!")
Hello Alice, Welcome!
                    # Using keyword arguments
```

Keyword arguments allow you to pass arguments to a function using the parameter name.
They make your code more **readable, flexible** and **reduce errors** in the order of arguments.

## 1. What are Keyword Arguments?

- Arguments passed to a function by specifying the parameter name.
- The order of arguments does not matter.
- Improves readability of the code.
- Must match the parameter name in the function definition.

## 2. Syntax

```
def function_name(param1, param2, param3):
    # function body

# Calling with keyword arguments
function_name(param2=value2, param1=value1,
              param3=value3)
```

## 3. Rules to Remember

1. The parameter name and spelling must match exactly.
2. Order of keyword arguments does not matter.
3. Each parameter can be given a value only once.
4. Keyword arguments must come after positional arguments.
5. You cannot use a keyword that is not a valid parameter name.

## 4. Invalid Examples

1. Wrong parameter name

```
def add(a, b):
    return a + b
print(add(x=10, y=20))   # Error: x, y not valid
```

2. Same parameter value twice

```
def add(a, b):
    return a + b
print(add(a=10, a=20))   # Error: multiple values
```

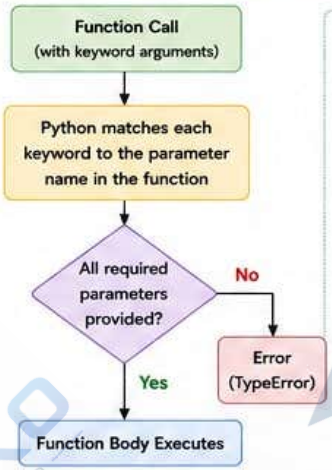
3. Keyword before positional

```
def add(a, b):
    return a + b
print(add(a=10, 20))     # Error: SyntaxError
```

## 5. Examples

| | Example Code | Explanation | Output |
|---|---|---|---|
| **5.1** Basic Example | `def greet(name, msg):`<br>`  print(f"Hello {name}, {msg}")`<br><br>`# Using keyword arguments`<br>`greet(name="Riya", msg="Have a nice day!")` | • Arguments are passed using parameter names.<br>• Order doesn't matter. | Hello Riya,<br>Have a nice day! |
| **5.2** Order Does Not Matter | `def info(name, age, city):`<br>`  print(name, age, city)`<br><br>`info(city="Delhi", name="Amit", age=21)` | • The arguments are in different order but still valid. | Amit 21 Delhi |
| **5.3** Mixing Positional and Keyword Arguments | `def power(base, exp, mod=None):`<br>`  if mod:`<br>`  return pow(base, exp, mod)`<br>`  return pow(base, exp)`<br><br>`print(power(2, exp=3))`<br>`print(power(2, 3, mod=5))` | • Positional arguments must come first.<br>• Remaining can be keyword arguments. | 8<br>3 |
| **5.4** With Default Parameters | `def greet(name, msg="Hello"):`<br>`  print(f"{msg}, {name}!")`<br><br>`greet(name="Sam")`<br>`greet(name="Sam", msg="Welcome")` | • msg has default value "Hello".<br>• Overridden using keyword. | Hello, Sam!<br>Welcome, Sam! |
| **5.5** All Keyword Arguments | `def rect(length, breadth):`<br>`  area = length * breadth`<br>`  return area`<br><br>`print(rect(length=10, breadth=5))` | • All arguments are passed as keywords. | 50 |

## 6. How Python Processes Keyword Arguments




### Key Points

- Python internally maps each keyword to the correct parameter.
- Missing any required parameter → Error.
- Extra/unknown keyword → Error.
- Default values are used when no value is provided.

## 7. Positional vs Keyword Arguments

| Feature | Positional Arguments | Keyword Arguments |
|---|---|---|
| How passed? | By position (order matters) | By name (order doesn't matter) |
| Readability | Less readable (in long list) | More readable |
| Flexibility | Less flexible | More flexible |
| Best for | Short arguments | Long argument lists (Optional / Some values) |

## 8. Real-life Analogy


In keyword arguments, you **clearly** tell what each value means!

## 9. Best Practices

- Use keyword arguments for better readability.
- Use them when a function has many optional parameters.
- Always use meaningful parameter names.
- Prefer keyword arguments for clarity in team projects.
- Avoid using keywords that shadow built-in names (like list, dict, id, etc.).

## 10. Common Mistakes

- Misspelling parameter names.
- Using keyword before positional argument.
- Passing the same value multiple times.
- Using unknown parameter names.

**Pro Tip** Use keyword arguments when the function has many parameters, especially when most of them have default values.

## 11. Key Takeaway

**Keyword arguments make your code more expressive, reduce bugs caused by wrong order, and are especially useful for functions with many or optional parameters.**

**Write clear code. Use keyword arguments!**

# *args and **kwargs in Python

```
def example(*args, **kwargs):
    print("args:", args)
    print("kwargs:", kwargs)

example(1, 2, 3, name="Alice", city="Delhi")
# args: (1, 2, 3)
# kwargs: {'name': 'Alice', 'city': 'Delhi'}
```

*args (Non-Keyword Variable Length Arguments) and **kwargs (Keyword Variable Length Arguments) allow a function to accept an arbitrary number of positional and keyword arguments.

## 1. What are *args and **kwargs?

***args** — Allows a function to accept any number of positional (non-keyword) arguments. They are received as a `tuple`.

****kwargs** — Allows a function to accept any number of keyword arguments. They are received as a `dictionary`.

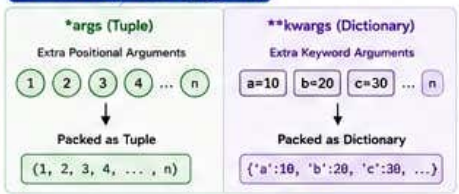
## 2. Syntax

```
def function_name(*args, **kwargs):
    # function body
```

## 3. Key Points

- `*args` collects extra positional arguments as a tuple.
- `**kwargs` collects extra keyword arguments as a dictionary.
- You can use either `*args` or `**kwargs` individually or both together.
- `*args` must come before `**kwargs` in the parameter list.
- You can still have normal parameters along with `*args` and `**kwargs`.

## 4. How They Work (Diagram)




## 5. Examples

| Example | Code | Explanation | Output |
|---|---|---|---|
| **5.1** Only *args | `def add_all(*args):`<br>`  total = 0`<br>`  for num in args:`<br>`  total += num`<br>`  return total`<br>`print(add_all(1, 2, 3))`<br>`print(add_all(10, 20, 30, 40, 50))` | • All positional arguments are packed into args (a tuple).<br>• We iterate over the tuple and add them. | 6<br>150 |
| **5.2** Only **kwargs | `def print_info(**kwargs):`<br>`  for key, value in kwargs.items():`<br>`  print(f"{key} : {value}")`<br>`print_info(name="Riya", age=21, city="Pune")` | • All keyword arguments are packed into kwargs (a dict).<br>• Access using keys. | name : Riya<br>age : 21<br>city : Pune |
| **5.3** Both *args and **kwargs | `def show(*args, **kwargs):`<br>`  print("args (tuple):", args)`<br>`  print("kwargs (dict):", kwargs)`<br>`show(10, 20, 30, name="Amit", age=22)` | • Positional args → args (tuple)<br>• Keyword args → kwargs (dict) | args (tuple): (10, 20, 30)<br>kwargs (dict): {'name': 'Amit', 'age': 22} |
| **5.4** With normal parameters | `def register(username, *args, **kwargs):`<br>`  print("Username:", username)`<br>`  print("Subjects:", args)`<br>`  print("Details:", kwargs)`<br>`register("John","Math","Science",age=20,city="Mumbai")` | • Normal params first<br>• Then *args<br>• Then **kwargs | Username: John<br>Subjects: ('Math', 'Science')<br>Details: {'age':20,'city':'Mumbai'} |
| **5.5** No extra arguments | `def greet(*args, **kwargs):`<br>`  print("Hello!")`<br>`  print("args:", args)`<br>`  print("kwargs:", kwargs)`<br>`greet()` | • args becomes empty tuple and kwargs becomes empty dictionary. | Hello!<br>args: ()<br>kwargs: {} |

## 6. Real-life Use Cases

### *args Use Cases

- Functions that work with any number of values (e.g., sum, max, min)
- Creating flexible APIs that accept variable inputs
- Logging functions that accept multiple messages

### **kwargs Use Cases

- Functions with many optional settings (e.g., configure app)
- Passing filters/conditions in database queries
- Building decorators or frameworks with options

## 7. *args/**kwargs vs Regular Arguments

| Feature | Regular Args | *args | **kwargs |
|---|---|---|---|
| Accepts how many? | Fixed number | Any number (positional) | Any number (keyword) |
| Stored as | Individual variables | Tuple | Dictionary |
| Access | By variable name | By index | By key |
| Order matter? | Yes | Yes (position matters) | No (order doesn't matter) |
| Use case | Known parameters | Variable positional inputs | Variable keyword options |

## 8. Common Mistakes

- Placing `*args` after `**kwargs` → `SyntaxError`
- Forgetting that `*args` is a tuple (not list).
- Forgetting that `**kwargs` is a dictionary (use `.items()`, `keys()`).
- Passing positional arguments after keyword arguments in function call.
  Wrong: `func(a=1, 2, 3)`
- Overusing `*args/**kwargs` when exact parameters are known (makes code less clear).

## 9. Important Rules (Remember!)

1. Normal parameters → `*args` → `**kwargs`
2. `*args` collects extra positional arguments into a tuple.
3. `**kwargs` collects extra keyword arguments into a dictionary.
4. They make functions more flexible and reusable!

## 10. Invalid Examples

```
def func(**kwargs, *args):
    pass      # SyntaxError

func(a=1, 2, 3)
    # Positional after keyword → Error

def func(*args, **kwargs, x): # invalid
    pass
```

## 11. Quick Summary

| | *args | **kwargs |
|---|---|---|
| Meaning | Any number of positional arguments | Any number of keyword arguments |
| Stored as | Tuple | Dictionary |
| Example call | `func(1, 2, 3, 4)` | `func(a=1, b=2, c=3)` |
| Example use | Sum numbers, math operations | Config options, settings, filters |

## 12. Best Practices

- Use `*args/**kwargs` for flexibility in reusable functions.
- Use meaningful names in `**kwargs` for clarity.
- Document your function to explain expected arguments.
- Don't overcomplicate—use only when needed!

**Pro Tip** — Think of `*args` as a bag for values and `**kwargs` as a box for labeled items.

**Flexible functions = Powerful code!**

Use *args and **kwargs wisely and write Pythonic code!

# COMMAND LINE ARGUMENTS iN PYTHON

Command line arguments are values passed to a Python program when it is run from the terminal (or command prompt).
They allow programs to be more flexible and user-friendly.

```
Terminal
$ python demo.py 10 20 Hello
          script name   command line arguments
```

CLI

## 1. What are Command Line Arguments?

- Values provided to a program at the time of execution.
- They are separated by spaces.
- In Python, we access them using the `sys` module.

## 2. How to Access?

We use the `sys` module.

```
import sys          # import module
print(sys.argv)     # list of arguments
```

`sys.argv` is a list that contains:

- `sys.argv[0]` → the script name
- `sys.argv[1]`, `sys.argv[2]`, ... → arguments

## 3. Example: Basic Usage

```
# demo.py
import sys
print("Total arguments:", len(sys.argv))
for i, arg in enumerate(sys.argv):
    print(f"argv[{i}] = {arg}")
```

Run in terminal:

```
$ python demo.py 10 20 Hello
```

Output:

```
Total arguments: 4
argv[0] = demo.py
argv[1] = 10
argv[2] = 20
argv[3] = Hello
```

## 4. Examples with sys.argv

| Example | Code | Run Command | Output |
|---|---|---|---|
| **4.1** Print all arguments | `import sys`<br>`print(sys.argv)` | `$ python demo.py`<br>`a b c` | `['demo.py',`<br>`'a', 'b', 'c']` |
| **4.2** Access individual arguments | `import sys`<br>`print("Name :"; sys.argv[1])`<br>`print("Age :"; sys.argv[2])` | `$ python demo.py`<br>`Alice 22` | `Name : Alice`<br>`Age : 22` |
| **4.3** Convert to specific type | `import sys`<br>`a = int(sys.argv[1])`<br>`b = float(sys.argv[2])`<br>`print("Sum =", a + b)` | `$ python demo.py`<br>`10 5.5` | `Sum = 15.5` |
| **4.4** All arguments (except script name) | `import sys`<br>`args = sys.argv[1:]`<br>`print(args)` | `$ python demo.py`<br>`x y z` | `['x', 'y', 'z']` |

★ Remember: `sys.argv[0]` is always the script name.

## 5. Working with Different Number of Arguments

```
import sys

if len(sys.argv) < 2:
    print("Usage: python demo.py <arg1> <arg2> ...")
    sys.exit(1)

print("You provided", len(sys.argv)-1, "argument(s):")
for arg in sys.argv[1:]:
    print("-", arg)
```

Run without arguments:

```
$ python demo.py
```

Output:

```
Usage: python demo.py
<arg1> <arg2> ...
```

Run with arguments:

```
$ python demo.py apple banana orange
```

Output:

```
You provided 3 argument(s):
- apple
- banana
- orange
```

## 6. Common Use Cases

- Configure programs without changing code (e.g., file paths, modes, limits)
- Process files (e.g., `input.txt output.txt`)
- Provide dynamic values (e.g., numbers, thresholds)
- Build flexible command line tools and automation scripts

## 7. Common Mistakes

- ✗ Forgetting that `argv[0]` is the script name.
- ✗ Accessing an index that doesn't exist → `IndexError`.
- ✗ Not converting argument types (e.g., using `int()`).
- ✗ Assuming arguments are always provided.
- ✗ Relying on the order without clear documentation.

## 8. Best Practices

- ✓ Always validate number of arguments.
- ✓ Provide helpful `usage` messages.
- ✓ Convert arguments to proper data types.
- ✓ Give meaningful names and documentation.
- ✓ Use modules like `argparse` for advanced parsing.

## 9. sys.argv Summary

| Index | Meaning | Example (`python demo.py 10 20 Hello`) |
|---|---|---|
| `sys.argv[0]` | Script name | `'demo.py'` |
| `sys.argv[1]` | First argument | `'10'` |
| `sys.argv[2]` | Second argument | `'20'` |
| `sys.argv[3]` | Third argument | `'Hello'` |
| `sys.argv[n]` | Nth argument | `'value'` |

## 10. Analogy

Think of command line arguments as values you hand to a program when asking it to do a task.

You → `$ python demo.py 10 20 Hello` (Give instructions (arguments)) → Program (`</>`) → Does the job

## 11. Key Takeaways

- ★ Command line arguments make programs flexible.
- ★ Access them using `sys.argv` list.
- ★ `argv[0]` is the script name; the rest are your arguments.
- ★ Validate, convert, and handle errors properly.
- ★ For complex needs, use `argparse` module.

**Pro Tip:** Use `argparse` module for named arguments, flags, and automatic help messages.
(Example: `python demo.py --name Alice --age 22`)

**Better programs start with better inputs!**

**Learn. Code. Automate.**
**Python makes it possible.**


1010101010
0101010101
1010101010
0101010101
</>
</>
Unit - IV
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

# STRINGS: Creating and Storing Strings in Python

A string in Python is a sequence of characters enclosed within quotes.
Strings are immutable, ordered and can contain letters, digits, symbols and spaces.

```
Example
s1 = 'Hello World'
s2 = "Python 3.12"
s3 = '''Multi
     Line String'''
```

Strings are used to store and work with textual data.

## 1. What is a String?

- A string is a collection of characters.
- Enclosed in single quotes (' '), double quotes (" ") or triple quotes ('' ''').
- Characters include letters, digits, symbols and spaces.
- In Python, strings are immutable (cannot be changed after creation).

## 2. Creating Strings

We can create strings using:

1. Single quotes
   `s1 = 'Hello'`
2. Double quotes
   `s2 = "Hello"`
3. Triple quotes (multi-line strings)
   ```
   s3 = '''This is a
        multi-line
        string'''
   ```
4. Empty string
   `s4 = ""` or `s5 = ''`

Note: Triple quotes are used for multi-line strings and docstrings.

## 3. Examples of Creating and Storing Strings

| # | Code | Description | Output / Value |
|---|---|---|---|
| 1 | `s = 'Python'` | String using single quotes | 'Python' |
| 2 | `s = "Python"` | String using double quotes | 'Python' |
| 3 | `s = 'Py thon 123'` | String with spaces and digits | 'Py thon 123' |
| 4 | `s = "Python's World"` | String with apostrophe | "Python's World" |
| 5 | `s = 'He said, "Hi!"'` | String with double quotes inside | 'He said, "Hi!"' |
| 6 | `s = """Hello`<br>`Welcome to Python`<br>`Enjoy Learning!"""` | Multi-line string | 'Hello<br>Welcome to Python<br>Enjoy Learning!' |
| 7 | `s = """` | Empty string | '' |
| 8 | `s = '12345'` | Digits stored as string | '12345'<br>(type is str, not int) |

## 4. How Strings are Stored in Memory

- Strings are stored as a sequence of Unicode characters.
- Each character occupies memory.
- The variable name (identifier) holds a reference to that string object in memory.


Each box represents a character.
\0 is the string terminator (internal representation).

## 5. Important Points

- Strings are immutable (cannot be changed).
- We cannot change individual characters of a string.
- Use indexing and slicing to access parts of a string.
- Use + to concatenate (join) strings.
- Use * to repeat strings.

```
Examples
s = 'Hi'
s + ' There'  → 'Hi There'
s * 3         → 'HiHiHi'
```

## 6. String vs Other Data Types

| Value | Type | Example |
|---|---|---|
| String | str | '123' (this is string) |
| Integer | int | 123 (this is integer) |
| Float | float | 12.3 |
| Boolean | bool | True or False |

Always remember: '123' ≠ 123
'123' is a string, 123 is an integer.

## 7. Common Operations on Strings (Basics)

```
Length        | len(s)          # number of characters
Concatenation | s1 + s2         # join strings
Repetition    | s * n           # repeat n times
Indexing      | s[i]            # character at index i
Slicing       | s[start:end]    # substring
```

Example

```
s1 = 'Hello'
s2 = 'World'
print(s1 + ' ' + s2)      # Hello World
print(len(s1))            # 5
print(s1 * 3)             # HelloHelloHello
print(s1[1])              # e
print(s1[1:4])            # ell
```

## 8. Common Mistakes

- Using single '=' instead of '==' while comparing.
  `if s = 'Hello':  # Wrong`
- Strings are immutable, so item assignment is invalid.
  `s[0] = 'h'  # TypeError`
- Forgetting that numbers inside quotes are strings.
  ```
  x = '10'
  print(type(x))   # <class 'str'>
  ```
- Mixing data types in operations.
  `'10' + 5  # TypeError`

Remember:
A string is text, treat it as text!

## 9. Key Takeaways

- Strings are created using '', "", or """ """.
- Strings are used to store textual data.
- Strings are immutable and ordered.
- We can perform many useful operations on strings.
- Always remember the difference between string and other data types.

Master strings, master text processing!
Practice more, code better!

**Pro Tip** Use meaningful variable names for strings.
`name = 'Alice'  # Good    x = 'Alice'  # Not meaningful`

**Practice:** Create strings, print them, and try different operations.

Practice today, build better programs tomorrow!

# BASIC STRING OPERATIONS in Python

A string in Python is a sequence of characters. Python provides a rich set of built-in operations that allow us to work with strings easily and efficiently.

```
Example:
s = "Hello, Python!"
print(len(s))        # 13
print(s.upper())     # HELLO, PYTHON!
```

## 1. What are Basic String Operations?

- Operations that allow us to access, modify, combine, or compare strings.
- Strings are immutable, so operations create new strings.
- Some common operations are: accessing, slicing, concatenation, repetition, membership, comparison, length, and case conversion.

## 2. Example String

```
s = "Hello, Python!"
```

| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | H | e | l | l | o | , | P | y | t | h | o | n | ! |

Length = 13

## 3. String Immutability

- We cannot change a string in place.
- Any operation that seems to modify a string actually returns a new string.

**Example:**

```
s = "Python"
s[0] = 'J'                # TypeError
s = s.replace('P', 'J')   # Correct
print(s)                  # Jython
```

## 4. Basic String Operations Overview

| Operation | Description |
|---|---|
| Length | Find number of characters |
| Concatenation | Join two or more strings |
| Repetition | Repeat a string multiple times |
| Indexing | Access a character by index |
| Slicing | Extract a part of the string |
| Membership | Check if substring exists |
| Comparison | Compare strings lexicographically |
| Case Conversion | Change case (upper/lower/etc.) |

## 5. Basic String Operations with Examples

| # | Operation | Syntax / Code | Example | Output / Result | Description |
|---|---|---|---|---|---|
| 1 | Length | `len(s)` | `s = "Hello"`<br>`len(s)` | 5 | Returns number of characters in string. |
| 2 | Concatenation (Joining) | `s1 + s2` | `s1 = "Hello"`<br>`s2 = " World"`<br>`s1 + s2` | 'Hello World' | Joins two or more strings. |
| 3 | Repetition | `s * n` | `s = "Hi"`<br>`s * 3` | 'HiHiHi' | Repeats the string n times. |
| 4 | Indexing | `s[index]` | `s = "Python"`<br>`s[0]`<br>`s[3]`<br>`s[-1]` | 'P'<br>'h'<br>'n' | Returns character at given index. Negative index counts from the end. |
| 5 | Slicing | `s[start:end:step]` | `s = "Python"`<br>`s[0:3]`<br>`s[2:5]`<br>`s[::2]` | 'Pyt'<br>'tho'<br>'Pto' | Returns substring from start to end-1 (step is optional). |
| 6 | Membership | `sub in s`<br>`sub not in s` | `s = "Python"`<br>`'Py' in s`<br>`'Java' not in s` | True<br>True | Checks if a substring exists in the string. |
| 7 | Comparison | `s1 == s2, s1 < s2, s1 > s2, etc.` | `'apple' == 'apple'`<br>`'apple' < 'banana'` | True<br>True | Compares strings lexicographically. |
| 8 | Case Conversion | `s.upper()`<br>`s.lower()`<br>`s.title()`<br>`s.swapcase()` | `s = "PyThOn"`<br>`s.upper()`<br>`s.lower()`<br>`s.title()`<br>`s.swapcase()` | 'PYTHON'<br>'python'<br>'Python'<br>'pYtHoN' | Changes case of characters. |

**Note:** All these operations return new strings. The original string remains unchanged.

## 6. More Useful String Operations

```
s.strip()              # remove leading/trailing spaces
s.lstrip()             # remove leading spaces
s.rstrip()             # remove trailing spaces
s.replace(old, new)    # replace substring
s.find(sub)            # first index of sub, -1 if not found
s.count(sub)           # number of occurrences
s.split(sep)           # split into list by separator
sep.join(list)         # join list into string using sep
```

| Example | Result |
|---|---|
| `" hi ".strip()` | 'hi' |
| `"banana".replace('a', 'o')` | 'bonono' |
| `"hello".find('e')` | 1 |
| `"hello".count('l')` | 2 |
| `"a,b,c".split(',')` | ['a', 'b', 'c'] |
| `"-".join(['a','b', "c"])` | 'a-b-c' |

## 7. Common Mistakes

- Trying to change a string character.
  ```
  s = "abc"    s[0] = 'z'    # TypeError
  ```
- Forgetting that slicing end index is exclusive.
  ```
  s = "hello"    s[1:4]    # 'ell', not 'ello'
  ```
- Using + for numbers with strings.
  ```
  "age:" + 10    # TypeError
  ```

## 8. Quick Practice

**Predict the output:**

1. `len("OpenAI")` = _____
2. `"Learn" + "ing"` = _____
3. `"Py" * 4` = _____
4. `"Python"[1]` = _____
5. `"Python"[1:4]` = _____
6. `"thon" in "Python"` = _____
7. `"apple" > "app"` = _____
8. `"hello".upper()` = _____

## 9. Real-life Applications

- Text processing and analysis
- Search and replace operations
- Form validation and user input handling
- Data cleaning and formatting
- Web development and parsing

## 10. Key Takeaways

- Strings are immutable sequences of characters.
- We can access, slice, join, repeat, and compare strings.
- Most operations return new strings.
- Mastering string operations is essential for solving problems efficiently.

**Strings are powerful — use them wisely!**

# ACCESSING CHARACTERS IN STRING BY INDEX NUMBER

In Python, each character in a string has a unique index number.
We use these index numbers to access individual characters from the string.

**Quick Example**

```
s = "Python"
print(s[0])   # P
print(s[3])   # h
print(s[-1])  # n
```

## 1. What is Indexing?

- Indexing means accessing a character of a string using its position number.
- Python uses zero-based indexing.
- The first character is at index `0`.

## 2. Syntax

```
string_name[index]
```

index can be:

- Positive (`0` to `n-1`) → from left to right
- Negative (`-1` to `-n`) → from right to left

## 3. Example String

Let's take a string:

```
s = "Python"
```

It has `6` characters (index `0` to `5`)

| Index (Positive) | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Character | P | y | t | h | o | n |
| Index (Negative) | -6 | -5 | -4 | -3 | -2 | -1 |

## 4. Accessing Characters with Examples

| Expression | Refers to | Result |
|---|---|---|
| s[0] | First character | 'P' |
| s[1] | Second character | 'y' |
| s[2] | Third character | 't' |
| s[3] | Fourth character | 'h' |
| s[4] | Fifth character | 'o' |
| s[5] | Sixth character | 'n' |
| s[-1] | Last character | 'n' |
| s[-2] | Second last character | 'o' |
| s[-6] | First character | 'P' |

## 5. Visual Explanation

s = "Python"

| Index (Positive) | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | P | y | t | h | o | n |
| Index (Negative) | -6 | -5 | -4 | -3 | -2 | -1 |

- We start counting from `0` at the left.
- Negative index `-1` means last character, `-2` means second last, and so on.

**Try these in Python:**

```
print(s[0])    # P
print(s[2])    # t
print(s[5])    # n
print(s[-1])   # n
print(s[-3])   # h
```

**Note:**

- Accessing an index outside the valid range causes an `IndexError`.
- For string of length n, valid positive index: `0` to `n-1`
- For negative index: `-n` to `-1`

## 6. More Examples

| String | Expression | Explanation | Result |
|---|---|---|---|
| s = "Hello" | s[0] | First character | 'H' |
| | s[4] | Last character | 'o' |
| | s[-1] | Last character | 'o' |
| | s[-5] | First character | 'H' |
| | s[1] | Second character | 'e' |
| s = "12345" | s[2] | Third character | '3' |
| | s[-2] | Second last character | '4' |

## 7. Out of Range Example

```
s = "Python"   # length = 6
print(s[6])    # IndexError
print(s[-7])   # IndexError
```

**Why Error?**

Because there is no index `6` (positive) or `-7` (negative) in this string.

## 8. Key Points

- Indexing helps to access a single character.
- Index always starts from `0`.
- Negative index counts from the end.
- Strings are immutable, so we can only access, not change, characters.
- Index out of range causes `IndexError`.

## 9. Common Mistakes

- Using index starting from 1. (Wrong: s[1] for first character)
- Using index greater than length-1.
- Using negative index less than -length.
- Confusing positive and negative indexes.

## 10. Practice Yourself

Given `s = "Computer"`

1. What is s[0]? _____
2. What is s[3]? _____
3. What is s[-1]? _____
4. What is s[-3]? _____
5. What is s[8]? _____

**Pro Tip** Use negative indexing when you want to access characters from the end of the string easily!

**Remember:** Indexing makes it easy to work with individual characters stored inside a string.

"Practice more, code better!"

# STRING SLICING AND JOINING in Python

Python strings are immutable sequences of characters. We can extract a part of a string using **SLICING** and combine strings using **JOINING**.

**Quick Example**

```
s = "Python Programming"
print(s[0:6])      # Python
print("-".join(["Learn", "Python"]))
                   # Learn-Python
```

## 1. What is String Slicing?

- Slicing means extracting a substring from a given string.
- We use the colon (:) operator.
- Syntax:

```
string[start : stop : step]
```

start → starting index (inclusive)
stop → ending index (exclusive)
step → step size (default is 1)

## 2. How Slicing Works

- The character at 'start' is included.
- The character at 'stop' is excluded.
- If step is positive → move left to right.
- If step is negative → move right to left.

## 3. Index Reference

Let's take a string:

```
s = "Python Programming"
```

Positive Index:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | y | t | h | o | n | | P | o | g | r | m | m | i | n | g |

Negative Index:

| -17 | -16 | -15 | -14 | -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Length of s = 17

## 4. String Slicing Examples (s = "Python Programming")

| Expression | Result | Explanation |
|---|---|---|
| s[0:6] | 'Python' | From index 0 to 5 (6 is excluded) |
| s[7:18] | 'Programming' | From index 7 to end |
| s[:6] | 'Python' | From start to index 5 |
| s[7:] | 'Programming' | From index 7 to end |
| s[:] | 'Python Programming' | Whole string |
| s[2:10] | 'thon Prog' | From index 2 to 9 |
| s[::2] | 'Pto rgamn' | Every 2nd character |
| s[1::2] | 'yh oPrormig' | Every 2nd character from index 1 |
| s[-11:-1] | 'Programming' | Negative indexing (from end) |
| s[::-1] | 'gnimmargorP nohtyP' | Reverse the string |
| s[-1::-1] | 'gnimmargorP nohtyP' | Another way to reverse |

**Tip:** Think of slicing like [start : stop : step]
(Include start, exclude stop, move by step)

## 5. What is String Joining?

- Joining means combining multiple strings into one string.
- We use the join() method.
- Syntax:

```
separator.join(iterable)
```

separator → the string placed between elements.
iterable → any iterable (list, tuple, etc.) of strings.

## 6. String Joining Examples

| Code | Result |
|---|---|
| ' '.join(['Hello', 'World']) | 'Hello World' |
| '-'.join(['Learn', 'Python', 'Today']) | 'Learn-Python-Today' |
| ','.join(['a', 'b', 'c']) | 'a,b,c' |
| ''.join(['P', 'y', 't', 'h', 'o', 'n']) | 'Python' |
| ' \| '.join(['Red', 'Green', 'Blue']) | 'Red \| Green \| Blue' |

### Notes on join()

- All elements in the iterable must be strings. Otherwise, TypeError occurs.
- The separator can be any string (even empty string).
- Very useful for formatting output, CSV creation, menu display, etc.

```
lst = ['10', '20', '30']
print(','.join(lst))       # 10,20,30
```

## 7. Real-Life Uses

- Formatting text output
- Creating CSV data
- Building menus or navigation
- Generating URLs or paths

## 8. Join with Non-String Values

We must convert non-string values first.

```
numbers = [1, 2, 3, 4]
# Wrong: ','.join(numbers)
# TypeError

# Correct:
result = ','.join(str(x) for x in numbers)
print(result)      # 1,2,3,4
```

## 9. Common Mistakes (Slicing)

- Forgetting that stop index is excluded.
  `s = "Python"  s[0:3]  # 'Pyt' not 'Pyth'`
- Using index out of range. `s[100]  # IndexError`
- Wrong step value (0 is invalid). `s[::0]  # ValueError`
- Confusing positive and negative indexing.

## 10. Practice Yourself

### A. Slicing (s = "Data Science")

1. s[0:4] = _____
2. s[5:] = _____
3. s[:4] = _____
4. s[::2] = _____
5. s[-6:-1] = _____
6. s[::-1] = _____

### B. Joining

1. Join ['Python', 'is', 'fun'] with ' ' → _____
2. Join ['One', 'Two', 'Three'] with '-' → _____
3. Join ['2024', '05', '25'] with '/' → _____
4. Join ['A', 'B', 'C'] with '' → _____
5. Convert [10, 20, 30] to strings and join with ',' → _____

## 11. Common Mistakes (Joining)

- Trying to join non-string values.
  `','.join([1, 2, 3])  # TypeError`
- Forgetting to convert numbers to strings.
  `','.join(str(x) for x in [1,2,3])  # Correct`
- Using join() on a non-iterable.
  `','.join(123)  # TypeError`

**Pro Tip** Use slicing to extract, join() to combine.
Practice both — they make text processing easy and powerful!

### Key Takeaways

- Slicing uses [start:stop:step]
- stop is excluded
- Negative indexing starts from end
- Join strings with separator.join(iterable)
- Convert non-strings to strings before joining

**Practice today, Master tomorrow!**