Python Overview 🐍

Some Resources -

BOOKS
Python Textbook By IITM (CS1002 - Programming in Python)
Python for Everybody. Author: Charles R. Severance. Publisher: Shroff Publishers. ISBN: 9789352136278

Playlist By IITM (CS1002 - Programming In Python - Foundation Level)
https://www.youtube.com/playlist?list=PLjE5A9cukiHpyTbAGQNWaWTYu813ee0PW

Extra
Python
Python By CS50 (David Malan)
Short Intro by Chatgpt

Full Course Playlist
Python Playlist
100 Days With Python By Code With Harry
Python By Apna College

OneShots
HIndi
One shot by CodeWIthHarry
Python by Apna College
Python Part 1 by Hitesh Sir Chai and code
Python part 2 by Hitesh

English
Python By David J Malan Sir CS50
Learn Python (Basic)
Intermediate Python (Imp)

Projects
Beginners Level Projects
Gaming Project
Basic to Advance Projects
12 Projects
Playlist

OPPS
https://youtu.be/LMbGZOU-dtA?si=M_f0bFDrLtbapzbz

DSA
https://youtu.be/pkYVOmU3MgA?si=BUBqWDCLZPptQmo1

Remember Choose Only one teacher and Blindly follow until you learn basics like Datatypes , variables , conditional statements function , File I/O , Exceptional handling , OOP E.T.C ……

Colab Book : Colab Book For Python Overview

Content Table

- Datatype
- Operators and Operands
- *Operators in Python*
- *Expression*
- *Order of Execution*
- *Type() Function and Type Casting*
- *Input() Function*

Chapter 3 : Conditional Expression

- *Comparison / Relational Operators in Python*
- *Logical Operators in Python*
- *Conditional Expressions in Python and Types - If , If-else , If-elif-else , Ternary Operator ……...*

Chapter 4 : Loops

- *Loops and Types of Loop - While , For*
- *Infinite loop in python*
- *range() function*
- *Loop Control - Break , Continue , Pass*
- *For loop with else ……...*

Chapter 5 : Strings

- *Accessing Characters in a String*
- *String Slicing*
- *String Concatenation and Repetition*
- *len() function*
- *in and not in*
- *Splitting and Joining String*
- *String Comparison*
- *String methods*
- *Escape Sequence Character*
- *Formatted Printing → sep and end Parameter in print() function*
- *Formatting Using f-strings*
- *Formatting Using Modulo (%) Operator*
- *Formatting Using Format Method*

Chapter 6 : List and Tuple

- *List*
- *Creating Lists*
- *Accessing List Element*
- *Slicing*

- *Modifying Lists*
- *Adding Element*
- *Removing Element*
- *Searching in lists*
- *Sorting and Reversing*
- *List Operations*
- *Iteration over lists*
- *List Comprehensions*
- *Matrix using list*
- *String and Lists*
- *Equality and Comparison*
- *List Function*
- *List Methods*
- *Tuple*

Chapter 0 : Introduction

(Not so Important , If you want you can skip)

What is Programming

Programming is the process of writing instructions (code) that a computer can understand and execute to perform specific tasks. These instructions are written using programming languages like Python, JavaScript, C++, and many others.

Key Aspects of Programming:

1. **Logic and Problem-Solving** – Breaking down problems into step-by-step instructions.
2. **Syntax and Semantics** – Writing code using the rules of a programming language.
3. **Algorithms and Data Structures** – Efficient ways to organize and manipulate data.
4. **Debugging** – Finding and fixing errors in code.
5. **Execution** – Running the code to see the results.

In other words , " Programming is a way to instruct the computer to perform various tasks."

What is Python?

Python is a high-level, interpreted programming language known for its simplicity and readability. It was designed to be easy to learn and use, making it a popular choice for beginners and experienced developers alike. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

FEATURES OF PYTHON

• Easy to understand = Less development time

• Free and open source

• High level language

• Portable: Works on Linux / Windows / Mac.

• Fun to work with!

Historical Background of Python

Python was created by **Guido van Rossum** in the late 1980s and was first released in **1991**. The main goal behind Python's development was to create a language that emphasized **code readability and simplicity**. Van Rossum was influenced by the ABC programming language, but he wanted to create a language that was more extensible and versatile.

Key Milestones in Python's History:

- **1989:** Guido van Rossum started developing Python at Centrum Wiskunde & Informatica (CWI) in the Netherlands.
- **1991:** Python 0.9.0 was released, featuring exception handling, functions, and core data types (list, dict, strings).
- **2000:** Python 2.0 was released, introducing list comprehensions and garbage collection.
- **2008:** Python 3.0 was released, which improved the language but was **not backward compatible** with Python 2.
- **2020:** Python 2 reached its **end of life**, and Python 3 became the standard.

Uses of Python

Python is widely used in various fields due to its versatility and ease of use. Some of its key applications include:

1. Web Development

- Python is used to build web applications using frameworks like **Django** and **Flask**.
- It helps create dynamic and scalable websites efficiently.

2. Data Science and Machine Learning

- Python is heavily used in **data analysis, visualization, and artificial intelligence**.
- Libraries like **Pandas, NumPy, Scikit-Learn, TensorFlow, and PyTorch** make it popular in **machine learning and deep learning**.

3. Automation and Scripting

- Python is great for writing scripts to **automate repetitive tasks**, such as file handling, email processing, and web scraping.
- Tools like **BeautifulSoup and Selenium** are used for web scraping.

4. Game Development

- Python is used to create games with frameworks like **Pygame**.

- Some popular games, like **EVE Online and Battlefield 2**, used Python for scripting.

5. Cybersecurity and Ethical Hacking

- Python is widely used in cybersecurity for writing penetration testing scripts and malware analysis.
- Tools like **Metasploit and Scapy** are built using Python.

6. Internet of Things (IoT)

- Python is used in IoT applications with **Raspberry Pi** to control hardware and sensors.

7. Embedded Systems

- Python is used in embedded systems and robotics, making it useful for hardware programming.

8. Finance and Trading

- Python is used in **algorithmic trading** and financial analysis.
- Libraries like **QuantLib** and **Zipline** help in financial modeling.

9. Artificial Intelligence and Chatbots

- Python powers AI applications, including **chatbots, voice recognition, and image processing**.

10. Mobile App Development

- Frameworks like **Kivy** and **BeeWare** allow Python to be used in **mobile app development**.

Why is Python Popular?

- **Simple and Readable Syntax**
- **Large Community Support**
- **Cross-Platform Compatibility**
- **Extensive Libraries and Frameworks**
- **Strong in AI and Data Science**

Set-Up

Setting up Python on your local machine is simple. Follow these steps based on your operating system:

1. Installing Python on Windows 🖥️

Step 1: Download Python

1. Go to the official Python website: https://www.python.org/downloads/
2. Click on the latest version for Windows.
3. Download the **Windows Installer (64-bit)**.

Step 2: Install Python

1. Run the downloaded **.exe** file.
2. **Important:** Check the box **"Add Python to PATH"** at the bottom.
3. Click **"Install Now"** and wait for the installation to complete.
4. Once installed, click **"Close"**.

Step 3: Verify Installation

3.Open **Command Prompt (CMD)**.

2.Type:

```
python --version
```

3. If installed correctly, it will display the installed Python version.

Some Useful Python Library

Category	Library	Purpose / Usage
General Development	`Requests`	Makes HTTP requests, API calls

	`BeautifulSoup`	Web scraping, parsing HTML/XML
	`Flask`	Lightweight web framework
	`Django`	Full-featured web framework
Data Science & Statistics	`NumPy`	Numerical computations, arrays
	`Pandas`	Data manipulation, CSV handling
	`Matplotlib`	Data visualization, charts
	`Seaborn`	Statistical data visualization
	`SciPy`	Advanced mathematical functions
	`Statsmodels`	Statistical modeling, hypothesis testing
Machine Learning & AI	`Scikit-Learn`	ML algorithms (classification, regression)
	`TensorFlow`	Deep learning, AI model training
	`PyTorch`	Neural networks, AI model training
Mathematics	`SymPy`	Symbolic mathematics, algebra
	`MPmath`	High-precision arithmetic calculations
Finance & Economics	`QuantLib`	Financial analytics, pricing models
	`ffn`	Portfolio performance, risk analysis
Automation & Scripting	`Selenium`	Browser automation, testing
	`PyAutoGUI`	GUI automation (clicking, typing)
	`OpenCV`	Image processing, computer vision
Cybersecurity & Hacking	`Scapy`	Network packet analysis, penetration testing

	`Cryptography`	Encryption, hashing, secure communication
Internet of Things (IoT)	`RPi.GPIO`	Raspberry Pi GPIO control
	`Paho MQTT`	IoT communication protocol

Chapter 01 : Module Comments and Pip

Chapter 1 Google Colab

Let's write our very first python program. Create a file called hello.py and paste the below code in it.

```
print("hello world")          # print is a function (more later)
```

Execute this file (.py file) by typing python hello.py and you will see Hello World printed on the screen.

PIP (Python Installer Package)

Pip is the package manager for python. It allows you to install, upgrade, and manage Python libraries and dependencies from the Python Package Index (PyPI).
You can use pip to install a module on your system.

Modules

In Python, a **module** is a file containing Python code, such as functions, classes, and variables, that can be imported and used in other Python programs. Modules help organize and reuse code efficiently.

Types of Modules in Python

1. **Built-in Modules** – Pre-installed with Python (e.g., `math`, `random`, `os`).
2. **User-Defined Modules** – Created by users to organize their code.
3. **External Modules** – Installed using package managers like `pip` (e.g., `numpy`, `pandas`, `tensorflow`, `flask etc`).

1. Built-in Modules

Python has many built-in modules that provide functionality for different tasks.

Example: Using the `math` module

```
import math
print(math.sqrt(16))   # Output: 4.0
print(math.pi)         # Output: 3.141592653589793
```

Example: Using the `random` module

```
import random
print(random.randint(1, 10))  # Output: Random number between 1 and 10
```

2. User-Defined Modules

You can create your own modules by saving Python code in a `.py` file.

Example:

Create a file `my_module.py`:

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

Now, import and use it in another script:

```
import my_module
print(my_module.greet("Batman"))  # Output: Hello, Batman!
```

3. External Modules

External modules are installed using `pip`.

Example: Installing and using `numpy`

1.
Install `numpy`:

```
pip install numpy
```

2.
Use it in Python:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)  # Output: [1 2 3 4 5]
```

USING PYTHON AS A CALCULATOR

We can use python as a calculator by typing “python” on the terminal.
This opens **REPL** or **Read Evaluate Print Loop.**

COMMENTS

Comments are used to write something which the programmer does not want to execute. They are ignored by the Python interpreter. In Python, comments are used to explain code and make it more readable. This can be used to mark author name, date etc.

TYPES OF COMMENTS

There are two types of comments in python.

1. Single Line Comments: To write a single line comment just add a ‘#’ at the start of the line.
Eg :

```
# This is a Single-Line Comment
```

2. Multi line Comments (Triple Quotes """ """ or ''' ''') : To write multi-line comments you can use ‘#’ at each line or you can use the multiline string (""" """ or ‘’’ ‘’’)

Eg :

```
 """This is an amazing
example of a Multiline
comment!"""
```

```
'''This is an amazing
example of a Multiline
comment! '''
```

Indentation

Indentation means adding **spaces or tabs** at the beginning of a line to show which code belongs to which block.

In Python, **indentation is mandatory**. It tells Python which lines of code belong **inside a condition, loop, function, class, etc.**

Where do you use indentation?

- After `if`, `elif`, `else`
- Inside loops (`for`, `while`)
- In functions (`def`)
- In classes (`class`)
- Basically, **after a colon (`:`)**

Dynamic Typing in Python

Dynamic Typing means **you don't need to declare the data type** of a variable when you create it.
Python **automatically detects the type** of the variable at runtime based on the value you assign.

Eg :

```
x = 10           # x is an integer

x = "Batman"     # now x is a string

x = 3.14         # now x is a float
```

The **type of variable x changes** depending on the value assigned.

You don't need to specify the type like you do in statically typed languages (like C, Java, etc.).

Multiple Assignment

Multiple assignment means assigning values to **multiple variables** in **a single line**.

Eg :

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

del Statement

The `del` keyword is used to delete a variable or an item from a list, dictionary, etc.

- **Deleting a Variable:**

```
x = 100

del x

# print(x) #→ ❌ Error: x is not defined
```

- **Deleting List Elements:**

```
nums = [10, 20, 30, 40]
del nums[2]
print(nums)  # [10, 20, 40]
```

🔹 Deleting Slices: reference from nums list

```
nums = [10, 20, 30, 40]
del nums[1:]    # deletes from index 1 to end
print(nums)     # [10]
```

🔹 Deleting Dictionary Keys:

```
person = {"name": "Batman", "age": 30}
del person["age"]
print(person)  # {'name': 'Batman'}
```

in Operator (Membership Operator) and not in Operator

The `in` operator is used to check if a **value exists in a sequence** (like string, list, tuple, etc.).

🔹 Syntax:

```
value in sequence
value not in sequence
```

Eg :

```
# With strings
print('a' in 'Batman')        # True
print('z' in 'Batman')        # False

# With lists
nums = [1, 2, 3]
print(2 in nums)              # True
print(5 in nums)              # False
```

Chaining Operators

Python allows you to chain **comparison operators** to check multiple conditions in a clean way.

```
x = 10
print(5 < x < 20)      # True → checks if x > 5 AND x < 20
print(1 == x < 100)    # False → 1 == x is False, so result is False
```

Virtual Environment in Python

A **virtual environment (venv)** is an isolated workspace for Python projects.

- Each project can have its **own dependencies (packages/libraries)** without interfering with others.

- It avoids the "dependency hell" problem where different projects require different versions of the same library.

Why Use Virtual Environment

1. **Isolation** → Each project uses its own packages.
2. **Version Management** → Different projects can use different versions of the same library.
3. **Portability** → You can share requirements via `requirements.txt`.
4. **Clean Environment** → Prevents global Python installation from being cluttered.

How to Create and Use Virtual Environment

1. Create Virtual Environment

```
python -m venv myenv
```

Here:

- `python` → Use your Python interpreter (sometimes `python3`).
- `-m venv` → Module to create virtual environments.
- `myenv` → Name of the environment folder (can be anything).

2. Activate Virtual Environment

On Windows (PowerShell or CMD):

```
myenv\Scripts\activate
```

On macOS/Linux:

```
source myenv/bin/activate
```

Deactivate Virtual Environment

```
deactivate
```

Chapter 02 : Variables and DataType

Chapter 2 Google Colab

Variables

Variables are containers that are used to store values . Variables in Python are defined by using assignment operator = .

🗒️ Syntax:

```
variable_name = value
```

Example :

```
name = "Batman"          # string variable
age = 25                 # integer variable
price = 99.99            # float variable
is_cool = True           # boolean variable
```

Rule For Naming Variable

1 - A variable name can contain alphabets, digits, and underscores.
2 - A variable name can only start with an alphabet and underscores.
3 - A variable name can't start with a digit.
4 - No space is allowed to be used inside a variable name.
5 - We can't use Reserved keywords as Variable name

These are following reserved keywords Which have special meaning in python

Python reserves 35 keywords:

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

DATA TYPES

In programming, a **datatype** (or **data type**) defines the **kind of data** a variable can hold. Primarily these are the following data types in Python:

A - Numerical Type

1. Integers
2. Floating point numbers
3. Complex ***(ignore)***

Example :

```
a = 10          # int
b = 5.5         # float
c = 3 + 4j      # complex
```

B - Text Type

1. Strings

Example :

```
name = "Batman"
Num_1 = "123"
```

C - Boolean Type

1. Booleans

Example :

```
is_false = False
is_true = True
```

Remember : Use True/False not true/false

D - Sequence Type

1. List
2. Tuple
3. Dictionary
4. Set

Example:

```
fruits = ["apple", "banana", "cherry"]  # list
point = (10, 20)                        # tuple
person = {"name": "Batman", "age": 35}      #dictionary    -> Key value
pairs
s = {1, 2, 3, 3}  # Output: {1, 2, 3} → No duplicates       # Set
```

E - None Type:

1. None

Example:

```
empty_num = None      # none type means nothing
```

Note - In Python everything is object

```
a = 1    # < class 'int' >
b = 5.22 # < class 'float' >
c = "Ayush" , "10" #  <class 'String' >
# d = True/False   <class 'Boolean'>
```

Operators and Operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called *operands*.

The operators `+`, `-`, `*`, `/`, and `**` perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

OPERATORS IN PYTHON

Following are some common operators in python:

1. Arithmetic operators : +, -, *, / etc.
2. Assignment / Shorthand operators : =, +=, -= etc.
3. Comparison / Relational operators : ==, >, >=, <, != etc.
4. Logical operators : and, or, not.

7. Arithmetic operators

+ -> Addition (int or float) , Concatenation (str)

- -> Subtraction

* -> Multiplication (int or float) , Repetition (str)

/ -> Division

// -> Floor Division (Integer division)

% -> Modulus operator (remainder after division)

** -> Exponentiation (power)

8.Relational operators(It will compare and give True or False)

> -> Greater than

< -> Less than

>= -> Greater than or equal

<= -> Less than or equal

== -> Equal

!= -> Not equal

9. Logical operators

   and -> Returns True if both LHS and RHS are true

   or -> Returns False if both LHS and RHS are false

   not -> Returns negation of given operand

Expression

An *expression* is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
17
x
x + 17
```

Order of Execution

When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*. For mathematical operators, Python follows mathematical convention. The acronym *PEMDAS* is a useful way to remember the rules:

- *P*arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.

- *E*xponentiation has the next highest precedence, so `2**1+1` is 3, not 4, and `3*1**3` is 3, not 27.
- *M*ultiplication and *D*ivision have the same precedence, which is higher than *A*ddition and *S*ubtraction, which also have the same precedence. So `2*3-1` is 5, not 4, and `6+4/2` is 8, not 5.
- Operators with the same precedence are evaluated from left to right. So the expression `5-3-1` is 1, not 3, because the `5-3` happens first and then `1` is subtracted from 2.

When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

Type() Function

type() function is used to find the data type of a given variable in python.

- **Syntax:**

```
type(object)
```

Eg :

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

y = 3.14
print(type(y))      # Output: <class 'float'>

z = "Hello"
print(type(z))      # Output: <class 'str'>

a = True
print(type(a))      # Output: <class 'bool'>

num = "123"          # string
print(type(num))     # <class 'str'>

num = int(num)       # type cast to int
print(type(num))     # <class 'int'>
```

Typecasting

Type casting (also known as **type conversion**) is the process of converting the **data type** of a value from one type to another in programming.In Python, there are two types of type casting:

🔹 1. Implicit Type Casting (Automatic)

Python **automatically converts** one data type to another when it makes sense.

Eg :

```
a = 10          # int
b = 3.5         # float
c = a + b       # float, because Python converts 'a' to float automatically

print(c)        # Output: 13.5
print(type(c))  # Output: <class 'float'>
```

🔹 2. Explicit Type Casting (Manual)

You **manually convert** data from one type to another using built-in functions:

Function	Converts to...
`int()`	Integer
`float()`	Float
`str()`	String
`bool()`	Boolean
`list()`	List
`tuple()`	Tuple

Eg :

1 -

```
x = "100"
y = int(x)        # Convert string to integer
```

```
print(y)          # Output: 100
print(type(y))    # Output: <class 'int'>
```

2 -

```
a = 5
b = float(a)      # Convert integer to float
print(b)          # Output: 5.0
```

3 -

```
n = 10
s = str(n)        # Convert integer to string
print(s)          # Output: '10'
print(type(s))    # Output: <class 'str'>
```

Note :

```
 x = "abc"
 y = int(x)  # ❌ Error: invalid literal for int()
Point to remember When we Convert Type to boolean
```

 # Int / Float to Boolean -------

 # Boolean value of any number which is int or float data type returns True
 # Only 0 (0.00) returns False

```
x_1 = bool(10)  #  → True      # boolean of +ve no is "True"
y_1 = bool(0)   # → False     # boolean of 0 is "false"
z_1 = bool(-10) # → True     # boolean of -ve no is "true"
Z_12 = bool(0.00) #    → False     # boolean of 0.00 is "false"
Y_12 = bool(2.77) # → True
```

 # String to Boolean —---

 # Every Non Empty String returns True , only Empty String Returns False .

```
f_1 = bool("ayush") #   →  True
f_2 = bool("18.675") #  → True
f_3 = bool("0.00")  # →  True    # Here 0.00 is String Data type not float
```

```
f_4 = bool("-17.687") # → True
f_5 = bool("") # → False      # Empty String
f_6 = bool(" ") #  → True     # space
```

Input() function

The **`input()` function** in Python is used to take input from the user. It waits for the user to type something and press Enter, then returns the input as a **string**.

🔹 Basic Syntax

```
input("Your message here")
```

Eg:

```
name = input("What's your name? ")
print("Hello, " + name + "!")



age = int(input("Enter your age: "))  # for integers
height = float(input("Enter your height in meters: "))  # for decimals
```

Chapter 3 : Conditional Expression

Chapter 3 Google Colab

A boolean expression is an expression that is either true or false.

Comparison / Relational Operators in Python

Comparison operators are used to **compare two values**. They return a **Boolean result**—either `True` or `False`—based on whether the comparison is correct.

Comparison operators check **how two values relate to each other**—like whether they are equal, not equal, greater, or smaller.

Relational Operators are used to evaluate conditions inside the if statements.

Operator	Description	Example	Result
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`7 > 3`	`True`
`<`	Less than	`3 < 7`	`True`
`>=`	Greater than or equal to	`5 >= 5`	`True`
`<=`	Less than or equal to	`4 <= 6`	`True`

Eg :

```
a = 10
b = 20

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

```
print(a < b)     # True
print(a > b)     # False
print(a <= b)    # True
print(a >= b)    # False
```

The Python symbols are different from the mathematical symbols for the same operations. A common error is to use a single equal sign (=) instead of a double equal sign (==).

Remember that = is an assignment operator and == is a comparison operator. There is no such thing as =< or =>.

Logical Operators in Python – Definition

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English

Logical operators are used to combine two or more conditions and return a Boolean value (True or False) based on the logic.

Logical operators help you make **decisions based on multiple conditions**. You can check if **all**, **at least one**, or **none** of the conditions are true.

In python logical operators operate on conditional statements. For Example:
*• **and** – true if both operands are true else false.*
*• **or** – true if at least one operand is true or else false.*
*• **not** – inverts true to false & false to true.*

Operator	Description	Example

`and`	Returns `True` if **both** conditions are true	`x > 5 and x < 10`
`or`	Returns `True` if **at least one** condition is true	`x > 5 or x < 3`
`not`	**Reverses** the result (True becomes False, and vice versa)	`not (x > 5)`

Eg :

```
x = 8

# AND operator
print(x > 5 and x < 10)    # True, both conditions are true

# OR operator
print(x > 5 or x < 3)      # True, at least one is true

# NOT operator
print(not(x > 5))     # False, because x > 5 is True, and not True = False
```

NOTE - Any nonzero number is interpreted as “true.”
for Example
>> 67 and True returns True

Conditional Expressions in Python

Conditional expressions are used to **make decisions** in your Python program. They allow you to execute different pieces of code **based on whether a condition is `True` or `False`**.

A **conditional expression** in Python evaluates a condition and returns one value if the condition is `True`, and another value if it’s `False`.

Why use Conditional Expressions?

- To control the **flow of your program**.

- To **respond to different situations** dynamically.
- To **perform logic-based operations**.

Types of Conditional Expressions

1. `if` Statement

Executes a block of code **only if** the condition is `True`.

Syntax :

```
if condition:
    # code to run if condition is True
```

Eg :

```
x = 10
if x > 5:
    print("x is greater than 5")
```

2. `if...else` Statement

Executes one block if the condition is `True`, and another block if it's `False`.

Syntax :

```
if condition:
    # code to run if condition is True
else:
    # code to run if condition is False
```

Eg :

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

3. `if...elif...else` Statement

Used to check **multiple conditions** one after another.

Syntax :

```
if condition1:

    # code if condition1 is True

elif condition2:

    # code if condition2 is True

elif condition3:

    # code if condition3 is True

...

else:

    # code if all above conditions are False
```

Eg:

```
x = 5
if x > 5:
    print("Greater than 5")
elif x == 5:
    print("Equal to 5")
else:
    print("Less than 5")
```

4. Ternary Operator (Conditional Expression in One Line)

Syntax:

```
value_if_true if condition else value_if_false
```

Eg :

```
age = 18
result = "Adult" if age >= 18 else "Minor"
print(result)  # Output: Adult
```

This is also called a **conditional expression** or **inline if-else**.

Pro Tip : Read About Nested if , Nested if-else and Nested if - elif - else and uses . (Given in code of Chapter 3 or otherwise ask with any AI tool , but 1st be Comfortable with if , if-else , if-elif-else)

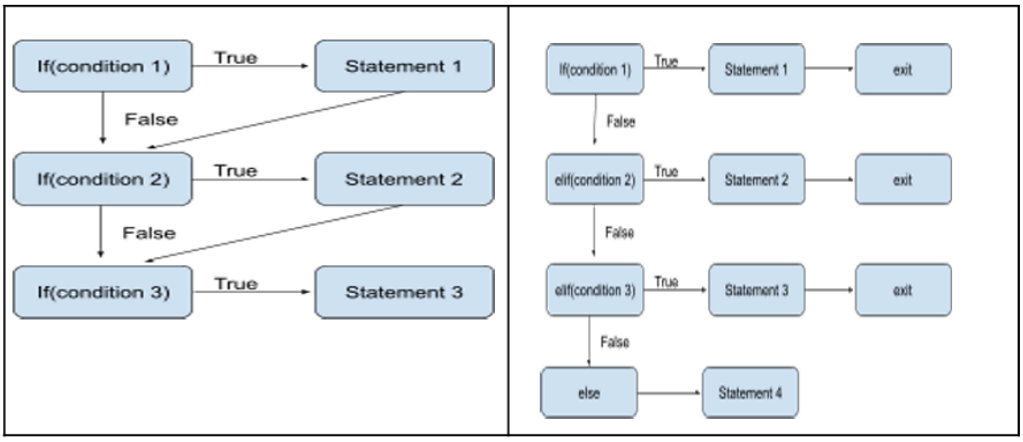
Summary Table

Statement Type	Description	Syntax Example
`if`	Check if condition is true	`if x > 5:`
`if...else`	Run code for true or false	`if x > 5: ... else: ...`
`if...elif...else`	Multiple condition checking	`if x > 5: ... elif x == 5: ... else: ...`
Ternary Operator	One-line condition result	`"Yes" if x > 5 else "No"`

IMPORTANT NOTES:

1. There can be any number of elif statements.
2. Last else is executed only if all the conditions inside elifs fail.

Flowchart:



Chapter 4 : Loops (Iteration)

Chapter 4 Google Colab

In **Python**, a **loop** is used to repeatedly execute a block of code as long as a condition is true or for a certain number of iterations.

Loops help avoid writing repetitive code.

Types of Loops in Python

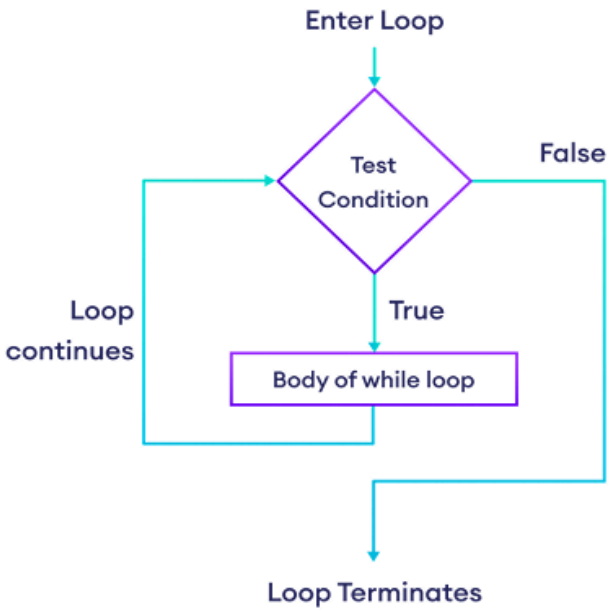
1. while loop

- Repeats a block of code as long as a condition is `True`.
- Good for unknown iterations.

Syntax :

```
while condition:

    # block of code

    # (repeats until condition becomes False)
```

Flowchart :



Example:

```
count = 0
while count < 5:
    print("Count:", count)
    count += 1
```

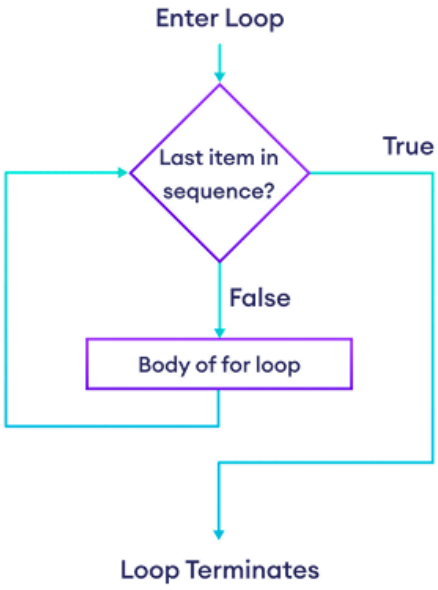
2. for loop

- Used to iterate over a sequence (list, tuple, string, range, etc.). (Iterable)
- Executes the block of code for each element in the sequence.
- Python's `for` loop is not like C/Java (where you write `for(i=0; i<n; i++)`).
- Instead, it **iterates directly over elements** of a collection.

Syntax :

```
for variable in sequence:

    # block of code

    # (executes for each item in the sequence)
```

Flowchart :



Example:

Example 1 – List Iteration

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

for fruit in fruits:

    print(fruit)
```

Example 2 – Range Iteration

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

    print(i)
```

Example 3 – String Iteration

```
for char in "Python":

    print(char)
```

Example 4 – Dictionary Iteration

```
student = {"name": "Ayush", "age": 20, "course": "Python"}

for key, value in student.items():

    print(key, ":", value)
```

Infinite Loop in Python

An infinite loop is a loop that never ends because the condition is always True (or never becomes False).

If not handled carefully, infinite loops can freeze your program.

1. Using while True

This is the most common way.

```
while True:
    print("This will run forever")

# Do not Run
# This will never stop until you break it or manually terminate (Ctrl + C
in terminal).
```

2. Using a condition that never changes

```
x = 5
while x > 0:  # condition is always true because x is not updated
    print("Infinite loop")
```

range() Function in Python

The range() function is used to generate a sequence of numbers.
It is often used in for loops when we want to repeat something a specific number of times.

Syntax

```
range(start, stop, step)
```

- **start** → (optional) starting number (default = 0)
- **stop** → (required) number where the sequence stops (exclusive)
- **step** → (optional) difference between each number (default = 1)

Example :

1. Only stop (default start=0, step=1)

```
for i in range(5):

    print(i)
```

2. Start and stop

```
for i in range(2, 7):

    print(i)
```

3. Start, stop, and step

```
for i in range(1, 10, 2):

    print(i)
```

4. Negative step (counting backwards)

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

    print(i)
```

Loop Control Statements

Sometimes you need to change the normal flow of loops:

- **break** → exits the loop immediately.
- **continue** → skips the current iteration and goes to the next.
- **pass** → does nothing (used as a placeholder).

1. break

- Exits the loop immediately.

Example:

```
for i in range(10):

    if i == 5:
```

```
        break

    print(i)
```

2. continue

- Skips the current iteration.

Example :

```
for i in range(5):

    if i == 2:

        continue

    print(i)
```

3. pass

- Placeholder (does nothing).

Example :

```
for i in range(5):
    pass   # useful when code is yet to be written
```

For Loop with Else :

1. Normal loop (no break)

```
for i in range(5):
    print(i)
else:
    print("Loop completed successfully")
```

Output :

```
0
1
2
```

```
3
4
Loop completed successfully
```

2. Loop with break

```
for i in range(5):

    if i == 3:

        break

    print(i)

else:

    print("Loop completed successfully")
```

Output :

```
0
1
2
```

Nested Loops in Python

A **nested loop** means placing one loop **inside another loop**.

- The **outer loop** runs first, and for **each iteration** of the outer loop, the **inner loop** runs completely.
- You can nest `for` inside `for`, `while` inside `while`, or mix `for` and `while`.

1. for loop inside for loop

```
for i in range(3):        # Outer loop

    for j in range(2):    # Inner loop

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

2. while loop inside while loop

```
i = 1

while i <= 3:       # Outer loop

    j = 1

    while j <= 2:  # Inner loop

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

        j += 1

    i += 1
```

3. for loop inside while loop

```
i = 0

while i < 3:              # Outer loop

    for j in range(2):    # Inner loop

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

    i += 1
```

4. while loop inside for loop

```
for i in range(3):    # Outer loop

    j = 0

    while j < 2:      # Inner loop
```

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

        j += 1
```

■ Basic difference between for loop and while loop

For Loop	While Loop
It is used when the number of iterations is known.	It is used when the number of iterations is not known.
It has a built-in loop control variable.	There is no built-in loop control variable.

■ Ideal loop option based on problem statement

Sr. No.	Problem statement	Ideal loop option
1	Find the factorial of the given number	for
2	Find the number of digits in the given number	while
3	Reverse the digits in the given number	while
4	Find whether the entered number is palindrome or not	while
5	Accept integers using input() function to find max until the input is -1	while
6	Print the multiplication table of the given number	for
7	Find whether the given number is prime or not	for
8	Find the sum of all digits in the given number	while
9	Find all positive numbers divisible by 3 or 5 which are than the given number smaller	for
10	Find all factors of the given number	for

Chapter 5 : Strings

Chapter 5 Google Colab

In Python, the `str` (string) data type is used to represent **textual data**. Strings are **sequences of characters** enclosed in **single (`'`)**, **double (`"`)**, or **triple quotes (`'''` or `"""`)**.

A **string** in Python is a **sequence of characters** enclosed within **single quotes `' '`, double quotes `" "`, or triple quotes `''' '''` / `""" """`**.
It is used to **represent textual data** such as words, sentences, or paragraphs.

Eg :

```
name = "Batman"  # Single-line String

greeting = 'Hello'       # Single-line String

paragraph = """This is a

multiline string."""  # Multi-line String
```

Facts :

- Strings are **immutable** (cannot be changed after creation).
- Strings are a type of **sequence**, so you can access characters using indexing and slicing.
- Python provides many built-in **string methods** for manipulation.
- In String , Indexing Starts from 0 .

Accessing Characters in a String

Strings are **indexed**, starting at **0**.

```
text = "Python"

print(text[0])    # 'P'

print(text[-1])   # 'n' (last character)
```

String Slicing

Syntax → str1[Start : End : Step] (Default Value for Start = 0 end = len(str1) : step =1)

```
s = "Hello World"
print(s[0:5])     # 'Hello'
print(s[:5])      # 'Hello'
print(s[6:])      # 'World'
print(s[::2])     # 'HloWrd' (every 2nd character)
```

String Concatenation & Repetition

```
a = "Hello"
b = "World"
print(a + " " + b)      # 'Hello World'       # Concatenation
print(a * 3)            # 'HelloHelloHello    #Repetition
```

```
num1 = "10"
num2 = "20"
print(num1+num2)
print(num1*3)
print(num1*int(num2))
```

Length of string by len() Function

len is a built-in function that returns the number of characters in a string.

```
s = "Python"
print(len(s))    # 6

s2 = "Hello World!"
print(len(s2))   # 12
```

Membership (`in`, `not in`)

- `in` and `not in` are **membership operators**.
- They are used to check if a **value exists** in a sequence (like string, list, tuple, set, dict).
- Return type: **Boolean** (`True` or `False`).

Example :

```
s = "Python"
print("Py" in s)    # True
print("Java" not in s)   # True
```

Splitting & Joining Strings

```
s = "apple,banana,cherry"
print(s.split(","))    # ['apple', 'banana', 'cherry']

words = ['I', 'love', 'Python']
print(" ".join(words))   # I love Python
```

String Comparison in Python

In Python, **strings can be compared** using **comparison operators** such as:

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<`	Less than

`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

Example:

```
a = "apple"
b = "banana"
c = "apple"
print(a == b)    # False
print(a == c)    # True
print(a != b)    # True
print(a < b)     # True (because 'a' comes before 'b' in alphabetical
order)
print(b > c)     # True
```

Important Notes:

- Python compares strings **character by character** using their **Unicode values (ASCII)**.
- Capital letters come **before** lowercase letters in Unicode.

```
print("Apple" < "apple")   # True
print("Zebra" > "apple")   # False
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters.

String Methods

Strings are an example of Python objects. An object contains both data (the actual string itself) and methods, which are effectively functions that are built into the object and are available to any instance of the object.

Python has a function called **dir** which lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods.

Eg :

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
[... 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(self, /)
    Return a capitalized version of the string.

    More specifically, make the first character have upper
    case and the rest lower case.
>>>
```

Imp String Methods

11. **String Methods**

Method	Description	Code x = 'pytHoN sTrIng mEthOdS'	Output
lower()	Converts a string into lower case	print(x.lower())	python string methods
upper()	Converts a string into upper case	print(x.upper())	PYTHON STRING METHODS
capitalize()	Converts the first character to upper case	print(x.capitalize())	Python string methods
title()	Converts the first character of each word to upper case	print(x.title())	Python String Methods
swapcase()	Swaps cases, lower case becomes upper case and vice versa	print(x.swapcase())	PYThOn StRiNG MeTHoDs

Method	Description	Code	Output
islower()	Returns True if all characters in the string are lower case	x = 'python' print(x.islower())	True
		x = 'Python' print(x.islower())	False
isupper()	Returns True if all characters in the string are upper case	x = 'PYTHON' print(x.isupper())	True
		x = 'PYTHoN' print(x.isupper())	False
istitle()	Returns True if the string follows the rules of a title	x = 'Pyhton String Methods' print(x.istitle())	True
		x = 'Pyhton string methods' print(x.istitle())	False

Method	Description	Code	Output
isdigit()	Returns True if all characters in the string are digits	x = '123' print(x.isdigit())	True
		x = '123abc' print(x.isdigit())	False
isalpha()	Returns True if all characters in the string are in alphabets	x = 'abc' print(x.isalpha())	True
		x = 'abc123' print(x.isalpha())	False
isalnum()	Returns True if all characters in the string are alpha-numeric	x = 'abc123' print(x.isalnum())	True
		x = 'abc123@*#' print(x.isalnum())	False

Method	Description	Code x = '-----Python-----'	Output
strip()	Returns a trimmed version of the string	print(x.strip('-'))	Python
lstrip()	Returns a left trim version of the string	print(x.lstrip('-'))	Python-----
rstrip()	Returns a right trim version of the string	print(x.rstrip('-'))	-----Python

Method	Description	Code x = 'Python'	Output
startswith()	Returns True if the string starts with the specified value	print(x.startswith('P'))	True
		print(x.startswith('p'))	False
endswith()	Returns True if the string ends with the specified value	print(x.endswith('n'))	True
		print(x.endswith('N'))	False

Method	Description	Code x = 'Python String Methods'	Output
count()	Returns the number of times a specified value occurs in a string	print(x.count('t'))	3
		print(x.count('s'))	1
index()	Searches the string for a specified value and returns the position of where it was found	print(x.index('t'))	2
		print(x.index('s'))	20
replace()	Returns a string where a specified value is replaced with a specified value	x = x.replace('S', 's') x = x.replace('M', 'm') print(x)	Python string methods

Note -

string.find(word) → This function searches a word and returns the index of the first occurrence of that word in the string.

string.replace (old word, new word) → This function replaces the old word with new word in the entire string.

Difference between String Function vs String methods

- **Functions** → general-purpose, external, called with `function(string)`
- **Methods** → string-specific, internal, called with `string.method()`

`Example :`

```
# Function

print(len(s))        # 18

# Method

print(s.upper())     # PYTHON PROGRAMMING
```

ESCAPE SEQUENCE CHARACTERS

What are Escape Sequence Characters?

Escape sequences are **special characters** in strings, used to represent things like newlines, tabs, quotes, etc. They start with a **backslash (`\`)**.

Escape Characters- To insert characters that are illegal in a string, use an escape character. An escape character is a backslash \ followed by the character you want to insert

Common Escape Sequence Characters

Escape Sequence	Meaning	Example Output
`\n`	New line	`"Hello\nWorld"` → Hello World
`\t`	Horizontal tab (4-8 spaces)	`"Hello\tWorld"` → Hello World
`\'`	Single quote	`'It\'s fine'` → It's fine
`\"`	Double quote	`"He said, \"Hi\""` → He said, "Hi"
`\\`	Backslash	`"C:\\Path\\file.txt"` → C:\Path\file.txt
`\r`	Carriage return (overwrites line)	`"123\rABC"` → ABC
`\b`	Backspace	`"abc\b"` → ab
`\f`	Form feed (page break, rarely used)	`"Hello\fWorld"`
`\a`	Bell sound (beep, may not work in all consoles)	`"\a"` (makes sound)
`\v`	Vertical tab (rarely used)	`"Hello\vWorld"`
`\ooo`	Character with octal value ooo	`"\101"` → 'A'
`\xhh`	Character with hex value hh	`"\x41"` → 'A'

Eg :

```
print("Line1\nLine2")
# Output:
# Line1
# Line2

print("Name\tAge")
# Output:
# Name    Age

print("He said, \"I am Batman\"")
# Output:
# He said, "I am Batman"

print('It\'s a trap!')
# Output:
# It's a trap!

print("C:\\Users\\Batman")
# Output:
# C:\Users\Batman
```

Tip:

To **avoid escape sequences** and treat backslashes as normal characters, use **raw strings** with `r""`:

```
print(r"C:\Users\Batman")   # Output: C:\Users\Batman
```

Formatted String Literals or f-strings

A formatted string literal (often referred to simply as an f-string) allows Python expressions to be used within string literals. This is accomplished by prepending an f to the string literal and enclosing expressions in curly braces {}

Eg :

```
years = 3

count = .1

species = "camels"

f"In {years} years I have spotted {count} {species}."

# In 3 years I have spotted 0.1 camels
```

■ Formatted Printing

sep parameter in print()

The `sep` parameter in the `print()` function is used to specify the **separator** between the items you are printing. By default, Python puts a **space** between multiple values in `print()`: , but by sep we can change it .

Syntax :

```
print(value1, value2, ..., sep="separator")
```

Eg -

```
print("2025", "04", "17", sep="-")
# output → 2025-04-17

print("Python", "is", "awesome", sep="🔥")
#output → Python🔥is🔥awesome
```

❖ **sep parameter in print():** The separator between the arguments to print() function in Python is space by default which can be modified and made to any character using the 'sep' parameter.

```
print("11", "06", "24")                  Output: 11 06 24
print("11", "06", "24", sep = "/")       Output: 11/06/24
print("11", "06", "24", sep = "S")       Output: 11S06S24
```

end parameter in print()

By **default**, every `print()` statement in Python ends with a **newline** (`\n`). That's why each `print()` call goes to the next line. The `end` parameter lets you **change what gets printed at the end** instead of the newline.

Syntax:

```
print(value1, value2, ..., end="your_custom_ending")
```

Eg:

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

❖ **end parameter in print():** By default Python's print() function ends with a newline(\n) which can be modified and made to any character using the 'end' parameter.

```
print("Hello")                           Output: Hello
print("python")                                  python

print("Hello", end = " ")                Output: Hello python. ^_^
print("python", end = ". ^_^ ")

print("Hello", "python", end = ".", sep = ", ")      Output: Hello, python.
```

Formatting using f-string

An f-string (formatted string literal) is a way to format strings in Python using expressions inside {}.
Introduced in Python 3.6, it's the fastest and most readable way to build strings with dynamic content.

Basic Syntax

```
f"some text {expression}"
```

Example:

```
name = "Batman"

print(f"Hello, {name}!")

# Output : Hello, Batman!
```

❖ **Formatting using f-string**

```
x = 5
print(f"value of x is {x}")          Output:  value of x is 5
print(f"value of x is {x:5d}")       Output:  value of x is     5
# Above print statement prints the value of the variable x with a field width of 5 characters, right-aligned. 'd' represents decimal number, 'f' represents float

pi = 22/7
print(f"value of pi is {pi}")        Output:  value of pi is 3.142857142857143
print(f"value of pi is {pi:.3f}")    Output:  value of pi is 3.143
# If we want a float number to nth decimal place we've to write '.nf'
print(f"value of pi is {pi:8.3f}")   Output:  value of pi is    3.143
```

Formatting using Modulo Operator (%)

String formatting using the % operator (also known as printf-style formatting) is a method in Python where format specifiers like %s, %d, and %f are used within a string to insert variables or values. It's an older formatting technique inspired by the C language's printf() function.

Although f-strings and .format() are more modern and preferred today, % formatting is still useful and you'll encounter it in older codebases.

Basic Syntax :

```
"format string" % (values)
```

Example :

```
name = "Batman"

print("Hello, %s!" % name)

# Output :  Hello, Batman!
```

Format Specifiers

Format	Meaning	Example
`%s`	String	`"Name: %s" % "Batman"`
`%d`	Integer (decimal)	`"Age: %d" % 30`
`%f`	Floating-point number	`"Pi: %.2f" % 3.1415`
`%x`	Hexadecimal (lower)	`"%x" % 255` → `ff`
`%o`	Octal	`"%o" % 10` → `12`
`%%`	Literal `%` character	`"Discount: 10%%"`

❖ **Formatting using Modulo Operator (%)**

```
x, pi = 5, 22/7

print("x = %d, pi = %f" % (x, pi))          Output:  x = 5, pi = 3.142857
# %d is replaced by 5 and %f is replaced by 22/7. Here, floating point number is printed up to 6 decimal places
```

```
print("x = %5d, pi = %.4f" % (x, pi))       Output:  x =     5, pi = 3.1429
#.4 is used to limit the decimal places of pi to 4
#5d, This specifies that the integer will take up at least 5 characters. If x has fewer digits, it will be padded with spaces on the left.

print("pi = %8.3f" % pi)                    Output:  pi =    3.143
#8 after mod, This specifies that the integer will take up at least 8 characters. If it has fewer digits, it will be padded with spaces on the left.
```

Formatting using Format Method

`.format()` Method in Python — Definition

The .format() method is a string formatting technique in Python that allows you to insert values into a string by placing curly braces {} as placeholders, which are replaced with specified arguments passed to .format().

Basic Syntax

```
"string with {} placeholders".format(values)
```

Example:

```
name = "Batman"
age = 35
print("My name is {} and I am {} years old.".format(name, age))

# Output: My name is Batman and I am 35 years old.
```

❖ Formatting using Format Method

```
pi = 22/7

print("x = {}, pi = {}".format(5, pi))          Output:  x = 5, pi = 3.142857142857143
#the first {} is given the first argument .i.e 5 and the second {} took the value or variable pi

print("x = {0}, pi = {1}".format(5, pi))        Output:  x = 5, pi = 3.142857142857143
print("pi = {1}, x = {0}".format(5, pi))        Output:  pi = 3.142857142857143, x = 5
# {0}, {1} represent the first and the second argument passed to the method.

print("x = {0:5d}, pi = {1:.4f}".format(5, pi))     Output:  x =     5, pi = 3.1429

print("pi = {0:8.3f}".format(pi))                   Output:  pi =    3.143
# In {0:8.3f}, '0' before ':' represents the argument no., as pi being 0th argument.
# '8' after ':' specifies that integer will take up at least 8 characters. If it has fewer digits, it will be padded with spaces on the left.
# '.3f' for the 3 decimal places

print("x = {x}, pi = {0:.4f}".format(pi, x = 5))    Output:  x = 5, pi = 3.1429
```

Chapter 6 : List and Tuple

[Chapter 6 Google Colab]

List

A **list** in Python is:

An ordered, mutable (changeable), and iterable collection of items that can store multiple values of any data type (integers, strings, floats, booleans, objects, or even other lists).

Key Properties of a List:

1. **Ordered** → Items have a defined sequence (indexed from `0` to `n-1`).
2. **Mutable** → You can modify, add, or remove elements after creating it.
3. **Heterogeneous** → Can hold mixed data types.
4. **Duplicates allowed** → Same value can appear multiple times.
5. **Dynamic size** → Can grow or shrink as needed.

Example:

```
my_list = [10, "Batman", 3.14, True, [1, 2, 3]]

print(my_list)         # [10, 'Batman', 3.14, True, [1, 2, 3]]

print(type(my_list))   # <class 'list'>
```

Creating Lists

You can create lists in different ways:

```
empty_list = []              # empty list
numbers = [1, 2, 3, 4, 5]    # list of integers
fruits = ["apple", "banana", "mango"]
mixed = [1, "hello", 3.14, True]

# Using list() constructor
chars = list("Python")       # ['P', 'y', 't', 'h', 'o', 'n']

# Nested list
```

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

Empty List → list() or [ ]

```
Empty_list = []
print(type(Empty_list))

Empty_list = list()
print(type(Empty_list))
```

Accessing List Elements

Lists are **indexed** (start from `0`).

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

print(fruits[0])   # apple

print(fruits[2])   # mango

# Negative indexing

print(fruits[-1])  # mango

print(fruits[-2])  # banana
```

Slicing → list[start : end : step]

```
numbers = [0, 1, 2, 3, 4, 5]

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

print(numbers[:3])    # [0, 1, 2]

print(numbers[3:])    # [3, 4, 5]

print(numbers[::-1])  # reversed → [5, 4, 3, 2, 1, 0]
```

Modifying Lists

Lists are **mutable**.

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

fruits[1] = "grapes"

print(fruits)  # ['apple', 'grapes', 'mango']

# Replacing multiple elements

fruits[0:2] = ["kiwi", "orange"]

print(fruits)  # ['kiwi', 'orange', 'mango']
```

Adding Elements

```
fruits = ["apple"]

fruits.append("banana")         # add at end
fruits.insert(1, "mango")       # insert at index
fruits.extend(["grapes", "kiwi"])  # add multiple elements

print(fruits)
# ['apple', 'mango', 'banana', 'grapes', 'kiwi']
```

Removing Elements

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

fruits.remove("banana")    # removes by value
print(fruits)  # ['apple', 'mango', 'grapes']

fruits.pop()     # removes last element
fruits.pop(0)    # removes by index
```

```
del fruits[0]    # delete specific index
del fruits       # delete whole list

fruits = ["apple", "banana", "mango"]
fruits.clear()   # empty list
print(fruits)    # []
```

Searching in Lists

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

print("banana" in fruits)    # True
print(fruits.index("apple"))  # 0 (first occurrence)
print(fruits.count("apple"))  # 2
```

Sorting and Reversing

```
numbers = [5, 3, 9, 1]
numbers.sort()         # [1, 3, 5, 9]
numbers.sort(reverse=True)  # [9, 5, 3, 1]

fruits = ["banana", "apple", "mango"]
print(sorted(fruits))    # ['apple', 'banana', 'mango'] (returns new list)

fruits.reverse()    # reverse order in-place
```

List Operations

```
a = [1, 2, 3]
b = [4, 5]

print(a + b)       # Concatenation → [1, 2, 3, 4, 5]
print(a * 3)       # Repetition → [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Iterating Over Lists

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

for fruit in fruits:
    print(fruit)

for i, fruit in enumerate(fruits):
    print(i, fruit)
```

List Comprehensions

A **list comprehension** is a concise way to create a list in a **single line** of code using a loop-like syntax.

It replaces:

```
result = []
for i in range(5):
    result.append(i**2)

print(result)
```

With :

```
result = [i**2 for i in range(5)]
print(result)
```

Basic Syntax : No condition

```
[expression for item in iterable]
```

expression → what to put in the list

item → variable representing each element

iterable → sequence (list, string, range, etc.)

Ex :

```
result = [i**2 for i in range(5)]
```

Basic Syntax : with if condition

```
[expression for item in iterable if condition]
```

Ex :

```
evens = [x for x in range(10) if x % 2 == 0]

print(evens)  # [0, 2, 4, 6, 8]
```

Basic Syntax : with if-else condition

```
[expression_if_true if condition else expression_if_false for item in
iterable]
```

Ex :

```
labels = ["even" if x % 2 == 0 else "odd" for x in range(5)]

print(labels)  # ['even', 'odd', 'even', 'odd', 'even']
```

Basic Syntax : With nested loops

```
[expression for item1 in iterable1 for item2 in iterable2 if condition]
```

Ex :

```
pairs = [(x, y) for x in [1, 2, 3] for y in [4, 5] if x != y]

print(pairs)  # [(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]
```

Matrix in Python using Lists

Since Python does not have a built-in matrix type (unlike NumPy), we typically represent a matrix as a **list of lists** (nested list).

```
matrix = [

    [1, 2, 3],

    [4, 5, 6],

    [7, 8, 9]

]

print(matrix)

# [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Accessing Elements and Updating Existing Element

```
# Like list it can also updated and accessed

matrix[0][1] = 99
print(matrix)
# [[1, 99, 3], [4, 5, 6], [7, 8, 9]]

print(matrix[0])       # First row → [1, 2, 3]
print(matrix[1][2])    # Row 2, Column 3 → 6
```

Conversion Between Strings & Lists

String → List

```
s = "Batman"
list1 = list(s)
print(list1)    # ['B', 'a', 't', 'm', 'a', 'n']

words = "I am Batman".split()
print(words)    # ['I', 'am', 'Batman']
```

List → String

```
l = ["I", "am", "Batman"]
sentence = " ".join(l)
print(sentence)    # I am Batman
```

Equality of Lists

Two lists are considered **equal (==)** if:

- They have the **same length**.
- Each element is **equal** and in the **same order**.

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

b = [1, 2, 3]

c = [3, 2, 1]

print(a == b)    # True  ✅ same elements in same order

print(a == c)    # False ❌ order matters

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

Identity vs Equality

`==` checks **value equality**,
while `is` checks **object identity** (whether both refer to the same memory location

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

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

c = a

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

print(a is b)  # False → different objects in memory

print(a is c)  # True → both point to the same object
```

Comparison of Lists (<, >, <=, >=)

Lists are compared **lexicographically** (like words in a dictionary):

- Compare element by element (left to right).
- First unequal element decides the result.
- If all elements are equal, shorter list is considered smaller.

Example 1 – Numeric lists

```
print([1, 2, 3] < [1, 2, 4])   # True (3 < 4)

print([1, 2, 3] > [1, 1, 5])   # True (2 > 1)

print([1, 2] < [1, 2, 0])      # True (shorter list is smaller if prefix
is equal)
```

Example 2 – Strings inside lists

```
print(["apple", "banana"] < ["apple", "cherry"])

# True → compare "banana" vs "cherry" → "b" < "c"
```

Popular List Function

Function	Description	Examples
len(list)	Returns the number of items in list.	a = [0, 1, 2, 3] #len(a) is 4
max(list)	Returns the largest item in the list.	a = [13, 32, -2, 5] #max(a) is 32
min(list)	Returns the smallest item in the list.	a = [13, 32, -2, 5] #min(a) is -2
sum(list)	Returns sum of all the elements of a list of numbers.	a = [13, 32, -2, 5] #sum(a) is 48
sorted(list, reverse = True/False)	Returns a new list with all elements of original list in: ● ascending order: (default) or if reverse = False ● Descending order: if reverse = True	#a = [13, 32, -2, 5] #sorted(a) = [-2, 5, 13, 32] or #sorted(a, reverse=False) = [-2, 5, 13, 32] #sorted(a, reverse=True) = [32, 13, 5, -2]
reversed(list)	Returns a list_reverse_iterator. *list(reversed(list)) can be used to get a new list containing all elements of original list in original order.	#a = [13, 32, -2, 5] #list(reversed(a)) = [5, -2, 32, 13] *note: do not forget to typecast reversed(a) to list datatype, as `reversed` doesn't return list data type.

Ex :

```python
# Python List Built-in Functions


lst = [10, 20, 30, 40, 50, 20, 0]

print(len(lst))         # 7 → length of list
print(max(lst))         # 50 → largest element
print(min(lst))         # 0 → smallest element
print(sum(lst))         # 170 → sum of elements
print(sorted(lst))      # [0, 10, 20, 20, 30, 40, 50] → returns new sorted
list
print(any(lst))         # True → at least one non-zero element exists
print(all(lst))         # False → because 0 is present (falsy)

# enumerate
for i, val in enumerate(lst):
    print(i, val)       # prints index with value

# list() constructor → converts other iterable into list
print(list((1, 2, 3)))      # [1, 2, 3]
print(list("Batman"))       # ['B','a','t','m','a','n']

# reversed
print(list(reversed(lst)))  # [0, 20, 50, 40, 30, 20, 10]
```

List Methods

Method	Description
`list.append(x)`	Adds an element to the end of the list. Equivalent to: a[len(a):] = [x]
`list.insert(i, x)`	Insert item `x` at index `i`.
`list.extend(iterable)`	Extend the list by appending all the elements of the iterable. Equivalent to: a[len(a):] = iterable
`list.remove(x)`	Removes first occurrence of element `x` from the list. Raises ValueError if `x` not in list.
`list.pop(i)`	Removes the item from index `i` and returns it. If no index specified then -1 is taken as default value for `i`.
`list.clear()`	Removes all elements from list. Equivalent to: del a[:]
`list.copy()`	Return a shallow copy of the list. Equivalent to: a[:]
`list.index(x)`	Return the index of first occurrence of `x`. Raises ValueError if `x` not in list.
`list.count(x)`	Return number of times `x` appeared in list. (0 if `x` not in list)
`list.sort(reverse = True/False)`	Sort items of list in place, both numerically and alphabetically. • Descending order if `reverse = True` • Ascending order if `reverse = False` (default parameter)
`list.reverse()`	Reverse the items of list in place.

Tuple

A **tuple** is an **ordered, immutable collection** of items in Python.

- **Ordered** → Elements have a defined order (like a list).
- **Immutable** → Once created, you cannot modify (add, remove, or change) its elements.
- Tuples are defined using **round brackets** `()`.

Creating Tuples

```
# Empty tuple

t1 = ()

# Tuple with integers

t2 = (1, 2, 3, 4)

# Tuple with mixed data types

t3 = (1, "Batman", 3.14, True)

# Nested tuple

t4 = (1, (2, 3), [4, 5])

# Tuple without parentheses (Python allows this)

t5 = 10, 20, 30

# Single element tuple (must add a comma!)

t6 = (5,)    # Tuple

not_tuple = (5)  # Just an integer

not_tuple2 = ('Ayush') # String
```

Empty Tuple

```
Empty_tuple = ()
print(type(Empty_tuple))

Empty_tuple = tuple()
print(type(Empty_tuple))
```

Accessing Tuple Elements

```
t = (10, 20, 30, 40, 50)

print(t[0])    # First element → 10

print(t[-1])   # Last element → 50

print(t[1:4]) # Slicing → (20, 30, 40)
```

Tuple Operations

```
# Tuple Operations
t1 = (1, 2, 3)
t2 = (4, 5, 6)

# Concatenation
print(t1 + t2)  # (1, 2, 3, 4, 5, 6)

# Repetition
print(t1 * 2)   # (1, 2, 3, 1, 2, 3)

# Membership
print(2 in t1)    # True
print(10 not in t1)  # True

# Length
print(len(t1))  # 3
```

Tuple Methods

Tuples only have two built-in methods:

```
t = (1, 2, 2, 3, 4, 2)
print(t.count(2))    # 3 → number of times 2 appears
print(t.index(3))    # 3 → index of first occurrence of 3
```

Immutability

```
t = (1, 2, 3)
# t[0] = 10   # ERROR → 'tuple' object does not support item assignment
```

But if a tuple contains a **mutable object** (like a list), that object can be modified:

```
t = (1, [2, 3], 4)
t[1][0] = 99
print(t)   # (1, [99, 3], 4)
```

Tuple Packing & Unpacking

```
# Packing
t = ("Bruce", "Wayne", "Batman")

# Unpacking
first, middle, last = t
print(first)   # Bruce
print(last)    # Batman
```

Tuple Methods

Method	Description	Example
tuple.count(x)	Returns no. of times `x` appeared in the tuple.	# t = (1, 2, 1, 4, 5, 6, 11) # t.count(1) -> 2
tuple.index(x)	Returns the index of first occurrence of `x` in tuple.	# t = (1, 2, 1, 4, 5, 6, 11) # t.index(1) -> 0

Tuple In Built Functions

```
# Tuple Built-in Functions

t = (10, 20, 30, 40, 50, 20)

print(len(t))        # 6 → length of tuple
print(max(t))        # 50 → largest element
print(min(t))        # 10 → smallest element
print(sum(t))        # 170 → sum of elements (only numbers)
print(sorted(t))     # [10, 20, 20, 30, 40, 50] → returns list
print(any(t))        # True → if any element is True/non-zero
print(all(t))        # True → if all elements are True/non-zero

# enumerate
for i, val in enumerate(t):
    print(i, val)    # prints index with value

# tuple() constructor → converts other data types into tuple
print(tuple([1, 2, 3]))     # (1, 2, 3)
print(tuple("Batman"))      # ('B','a','t','m','a','n')

# reversed
print(tuple(reversed(t)))   # (20, 50, 40, 30, 20, 10)
```

Note : Tuple Comparison , Slicing of Tuple , Tuple Concatenation and Replication and in operator is used similar like lists as we see before

We will Discuss Set and Dictionary in next Chapter

Quick Summary of Collections

Data Type	Iterable	Collection	Indexable/ Sliciable	Ordered	Mutable	Uses Hashing
str	✅	❌	✅	✅	❌	❌
range	✅	❌	❌	✅	❌	❌
list	✅	✅	✅	✅	✅	❌
tuple	✅	✅	✅	✅	❌	❌
set	✅	✅	❌	❌	✅	✅

Comprehensions

Just a short PS . we have discussed it earlier……

1. **Using a single line `if` statement:**

Syntax:

```
if condition:          ->      if condition: block
    block
```

Eg.

```
a = int(input())       ->    a = int(input())
if a%2 == 0:                 if a%2 == 0: print('even')
    print('even')
```

2. Using a single line `if- else` statement.

Syntax:

```
if condition:
    block1          ->      block1 if condition else block2
else:
    block2
```

Eg.

```
a = int(input())
if a%2 == 0:          ->    a = int(input())
    print('even')           print('even') if a%2 == 0 else print('odd')
else:
    print('odd')
```

3 . Using single line `for` loop:

Syntax:

```
for x in iterable:    ->    for x in iterable: line1; line2
    line1
    line2
```

Eg.

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

4. Using single line `while` loop:

Syntax:

```
while condition:    ->     while condition: line1; line2
    line1
    line2
```

Eg.

```
i = 0                          i = 0
while i<5:                ->   while i<5: print(i); i += 1
    print(i)
    i += 1
```

List Comprehensions

1. Mapping a list:

Syntax:

```
L = [ ]                    ->  L = [f(x) for x in iterable]
for x in iterable:
    L.append(f(x))
```

Eg. Create a new list whose elements are square of that of L1.

```
L1 = [1, 2, 3, 4]          ->   L1 = [1, 2, 3, 4]
L = [ ]                         L = [x**2 for x in L1]
for x in L1:
    L.append(x**2)
```

2. Filtering and Mapping (only using if with comprehension):

Syntax:

```
L = [ ]                 ->   L = [f(x) for x in iterable if condition]
for x in iterable:
    if condition:
        L.append(f(x))
```

Eg. Create a new list with only even elements of L1.

```
L = [ ]                 ->   L = [x for x in L1 if x%2 == 0]
for x in L1:
     if x%2 == 0:
          L.append(x)
```

3. if- else in List Comprehensions:

Syntax:

```
L = [ ]
for x in iterable:
     if condition:
          L.append(f(x))
     else:
          L.append(g(x))
```

```
->  L = [f(x) if condition else g(x) for x in iterable]
```

Eg. Create a new list by doubling the elements at even indices and squaring the elements at odd indices.

```
L = [ ]
for i in range(len(L1)):
     if i%2 == 0:
          L.append(L1[i]**2)
     else:
          L.append(L1[i]*2)
```

```
->   L = [L[i]**2 if i%2 == 0 else L[i]*2 for i in range(len(L1))]
```

Chapter 7 : Dictionary and Sets

[Chapter 7 Google Colab]

Dictionary

A **dictionary** in Python is a **mutable, unordered collection of items** that stores data in the form of **key–value pairs**, where:

- Each **key** is unique, immutable (such as strings, numbers, or tuples).
- Each **value** can be of any data type (mutable or immutable)

Syntax :

```
dict_name = { key1: value1, key2: value2, ... }
```

Characteristics :

1. Key-Value pairs – Each item has a unique key.
2. Keys must be immutable (string, number, tuple).
3. Values can be anything (int, string, list, dict, etc).
4. Unordered (no fixed position, but since Python 3.7+, insertion order is preserved).
5. Mutable – We can change values, add or remove items.

Example:

```
student = {

    "name": "Ayush",

    "age": 20,

    "is_student": True,

    "subjects": ["Python", "Math", "DSA"]

}

print(student)

# Output: {'name': 'Ayush', 'age': 20, 'is_student': True, 'subjects':
['Python', 'Math', 'DSA']}
```

Empty Dictionary

```
# Creating empty dictionary using {}
empty_dict1 = {}
print(empty_dict1, type(empty_dict1))  # {} <class 'dict'>

# Creating empty dictionary using dict()
empty_dict2 = dict()
print(empty_dict2, type(empty_dict2))  # {} <class 'dict'>
```

Accessing Dictionary Elements

```
student = {"name": "Ayush", "age": 20}

print(student["name"])      # Ayush (Direct access)
print(student.get("age"))   # 20 (Safely get value)

# If key not present
print(student.get("grade", "Not Found"))  # Output: Not Found
print(student["grade"])       # This will give error as key is not present
```

Modifying Dictionary

```
student = {"name": "Ayush", "age": 20}

# Change value
student["age"] = 21

# Add new key-value
student["course"] = "Python"

# Delete key
del student["age"]

# Remove and return a value
grade = student.pop("course")

print(student)      # {'name': 'Ayush'}
```

Dictionary Functions / Methods

```
d = {"a": 1, "b": 2, "c": 3}

print(len(d))        # 3 → Number of items
print(d.keys())      # dict_keys(['a', 'b', 'c'])
print(d.values())    # dict_values([1, 2, 3])
print(d.items())     # dict_items([('a', 1), ('b', 2), ('c', 3)])

# Loop through dictionary
for k, v in d.items():
    print(k, ":", v)

# Copy dictionary
d2 = d.copy()
print(d2)

# Clear dictionary
d.clear()
print(d)    # {}
```

Nesting Dictionary

```
students = {
    "student1": {"name": "Ayush", "age": 20},
    "student2": {"name": "Bruce", "age": 22}
}

print(students["student1"]["name"])
# Output: Ayush
```

Dictionary Comprehension

```
squares = {x: x**2 for x in range(5)}
print(squares)
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
# Create dictionary using comprehension
squares = {x: x**2 for x in range(5)}
print(squares)  # {0:0, 1:1, 2:4, 3:9, 4:16}

# Conditional comprehension
even_squares = {x: x**2 for x in range(10) if x % 2 == 0}
print(even_squares)
```

● Dictionaries in Python is a data structure that stores values in key: value format.

● A dictionary can be created by placing a sequence of elements within curly {} braces, separated by a ‘comma’.

● A dictionary can also be created by the built-in function dict()

```
# creating an empty dictionary

Dict = {}

print(type(Dict))

Dict = dict()

print(type(Dict))

# creating a dictionary in various ways

Dict = { 1: 'Python', 2: 'dictionary', 3: 'example' }

Dict = dict({1: 'Python', 2: 'dictionary', 3: 'example'})

Dict = dict([(1, 'Python'), (2, 'dictionary'), (3, 'example')])
```

● Values in a dictionary can be of any data type and can be duplicated, whereas keys can’t be repeated.

● Dictionary keys are case-sensitive, the same name but different cases of Key will be treated distinctly

```
Dict = { 1: 'Hello', 2: 123, 3: [32, 43, 12], 4: 123 }
```

```
# here values are different data types and also can be repeated

Dict = { 'py': 123, 'Py': 1234 }

# here, ‘py’ and ‘Py’ both are keys as keys are case-sensitive

Dict = { 1: 'Python', 1: 'dictionary', 3: 'example' }

print(Dict)    # {1: 'dictionary', 3: 'example'}

# while initialising a dictionary with same key, it always stores the
latest value
```

Adding & Updating Elements to a Dictionary

● Onevalueatatime can be added to aDictionary by defining value along with the key e.g. **Dict[Key] = ‘Value’.**

● While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.

● Updating an existing value in a Dictionary can be done by using the built-in update() method. Nested key values can also be added to an existing Dictionary

```
Dict = {} # Empty Dictionary

Dict[0] = 'Hello'

Dict[1] = 2

Dict['Value_set'] = 2, 3, 4

print(Dict)    # {0: 'Hello', 1: 2, 'Value_set': (2, 3, 4)}

Dict[1] = 'Changed'     # Updated key value

Dict.update({0:5})      # Update using update method

print(Dict)         # {0: 5, 1: 'Changed', 'Value_set': (2, 3, 4)}
```

Accessing and Deleting Elements to a Dictionary

● We can access the dictionary items by using keys inside square brackets.

● There is also a method called **get()** to access the dictionary items. This method accepts the key as an argument and returns the value.

● The items of the dictionary can be deleted by using the **del** keyword

```
Dict = {1: 'Hello', 'name': 'Python', 3: 'World'}



# Accessing an element using key

print(Dict['name'])      # output: Python



# Accessing an element using get method

print(Dict.get(3))       # output: World

del(Dict[1])             # delete an element

print(Dict)              # output: {'name': 'Python', 3: 'World'}
```

Nested Dictionary

A dictionary can be stored as the value of another dictionary

```
Dict = {1: 'IIT', 2: 'Madras', 3: {'A': 'Welcome', 'B': 'To', 'C':
'Python'}}

print(Dict)

# {1: 'IIT', 2: 'Madras', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Python'}}

# accessing element of nested dictionary
```

```
print(Dict[3]['A'])          # output: Welcome

# updating & adding element of nested dictionary

Dict[3]['B'] = "Into" # updating

Dict[3]['D'] = "World" # adding

print(Dict)

# {1: 'IIT', 2: 'Madras', 3: {'A': 'Welcome', 'B': 'Into', 'C': 'Python',
'D': 'World'}}

# accessing element of nested dictionary

del(Dict[3]['D'])

print(Dict)

# {1: 'IIT', 2: 'Madras', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Python'}}
```

Dictionary Methods

Method	Description
clear()	Removes all items from the dictionary
copy()	Returns a shallow copy of the dictionary
get(key, default = value)	Returns the value of the specified key. If the key is not present in the dictionary it returns the default value if any default value is passed
items()	Returns a list containing a tuple for each key-value pair
keys()	Returns a list containing the dictionary's keys
values()	Returns a list of all the values of dictionary

pop()	Remove the element with specified key
popitem()	Removes the last inserted key-value pair from the dictionary and returns it as a tuple
fromkeys()	Creates a new dictionary from the given sequence with the specific value
setdefault()	Returns the value of a key if the key is in the dictionary else inserts the key with a value to the dictionary
update()	Updates the dictionary with the elements from another dictionary or an iterable of key-value pairs. With this method, you can include new data or merge it with existing dictionary entries

Examples of important Dictionary Methods

```
d = {1: '001', 2: '010', 3: '011'}

print(d.get(2, "Not found"))        # 010, as 2 is a key
print(d.get(4, "Not found"))        # Not found, as 4 not present
print(d.items())     # dict_items([(1, '001'), (2, '010'), (3, '011')])
print(d.keys())      # dict_keys([1, 2, 3])
print(d.values())    # dict_values(['001', '010', '011'])
print(d.pop(1))      # 001
print(d)             # {2: '010', 3: '011'}
```

Sets

A **set** in Python is a **built-in data type** that represents an **unordered, mutable collection of unique, immutable, and hashable elements**.

- **Unordered** → Elements have no fixed position or index.
- **Mutable** → The set itself can be changed (add/remove elements).
- **Unique** → Duplicate elements are automatically removed.
- **Immutable & Hashable elements** → Only immutable objects (e.g., numbers, strings, tuples) can be stored in a set.

Ex :

```
# Empty set (Remember not {} , This is dictionary)
s = set()

# Set with elements
s1 = {1, 2, 3, 4}
s2 = set([1, 2, 2, 3, 4])  # duplicates removed

print(s1)  # {1, 2, 3, 4}
print(s2)  # {1, 2, 3, 4}
```

Valid Elements of Sets:
● Any immutable and hashable objects can be used as elements of sets. Eg. int, tuple, string.
● Mutable objects like lists, sets and dictionaries cannot be used as elements of list.
● Note → item = (1, 2, [3, 4], 5) Here a is a tuple but still non-hashable as it contains a list which is mutable. Hence `item` can't be an element of a set.

Set Operations :

- Let's take set1 = {1, 2, 3, 4} and set2 = {1, 3, 5}

Operation	Expression	Example
Union	`set1 | set2`	{1, 2, 3, 4, 5}
Intersection	`set1 & set2`	{1, 3}
Set Difference	`set1 - set2`	{2, 4}
Symmetric Difference	`set1 ^ set2`	{2, 4, 5}
Subset	`set1 <= set2` (Is set1 subset of set2?)	False #eg. {1, 2, 3} <= {1, 2, 3} : True
Proper Subset	`set1 < set2`	False #eg. {1, 2} < {1, 2, 3} : True
Superset	`set1 >= set2` (Is set1 a superset of set2?)	False #eg. {1, 2, 3} >= {1, 2, 3} : True
Proper Superset	`set1 > set2`	False #eg. {1, 2} > {1} : True

Ex :

```
# Define the sets
set1 = {1, 2, 3, 4}
set2 = {1, 3, 5}

# Union
union_set = set1 | set2
print("Union:", union_set)  # {1, 2, 3, 4, 5}

# Intersection
intersection_set = set1 & set2
print("Intersection:", intersection_set)  # {1, 3}

# Set Difference
difference_set = set1 - set2
print("Set Difference (set1 - set2):", difference_set)  # {2, 4}
```

```
# Symmetric Difference
sym_diff_set = set1 ^ set2
print("Symmetric Difference:", sym_diff_set)  # {2, 4, 5}

# Subset
is_subset = set1 <= set2
print("Is set1 subset of set2?:", is_subset)  # False

# Proper Subset
is_proper_subset = set1 < set2
print("Is set1 proper subset of set2?:", is_proper_subset)  # False

# Superset
is_superset = set1 >= set2
print("Is set1 superset of set2?:", is_superset)  # False

# Proper Superset
is_proper_superset = set1 > set2
print("Is set1 proper superset of set2?:", is_proper_superset)  # False

# Additional examples to show True cases
print("\n# True cases examples:")
print("{1,2,3} <= {1,2,3}:", {1,2,3} <= {1,2,3})  # True subset
print("{1,2} < {1,2,3}:", {1,2} < {1,2,3})        # True proper subset
print("{1,2,3} >= {1,2}:", {1,2,3} >= {1,2})      # True superset
print("{1,2,3} > {1,2}:", {1,2,3} > {1,2})        # True proper superset
```

Set Methods :

Methods	Description
`set.add(x)`	Adds element `x` to the set.

`set.update(iterable)`	Adds all the elements of the `iterable` to set.
`set.remove(x)`	Removes element `x` from set. Raises error if element not found.
`set.discard(x)`	Removes element `x` from set. Do not raise any error if element not found.
`set.pop()`	Removes and returns a random element from set. Error in case of empty set.
`set.clear()`	Removes all elements from set.
`set.copy()`	Returns a shallow copy of set.
`set1.union(set2)`	Returns a new set which is union of set1 and set2.
`set1.intersection(set2)`	Returns a new set which is intersection of set1 and set2.
`set1.difference(set2)`	Returns a new set which is equivalent to set1 - set2
`set1.difference_update(set2)`	Removes all the elements of set1 which are present in set2.
`set1.symmetric_difference(set2)`	Returns a new set which is equivalent to set1 ^ set2
`set1.symmetric_difference_update(set2)`	Update the set1 keeping the elements found in either of set1 and set2, but not in both set1 and set2
`set1.issubset(set2)`	Returns `True` if set1 is subset of set2
`set1.issuperset(set2)`	Returns `True` is set1 is superset of set2
`set1.isdisjoint(set2)`	Returns `True` if both the set has no common elements.

Ex :

```
# Example of set methods in Python

# Creating a set
my_set = {1, 2, 3}
```

```python
# Adding an element
my_set.add(4)  # Adds 4 to the set
print(my_set)  # Output: {1, 2, 3, 4}

# Adding a duplicate element
my_set.add(2)  # 2 is already in the set, so nothing changes
print(my_set)  # Output: {1, 2, 3, 4}


# 1. update()
s = {1, 2}
s.update([3, 4, 5])
print("update:", s)  # Output: {1, 2, 3, 4, 5}

# 2. remove()
s.remove(3)
print("remove:", s)  # Output: {1, 2, 4, 5}
# s.remove(10)  # Would raise KeyError

# 3. discard()
s.discard(4)
print("discard:", s)  # Output: {1, 2, 5}
s.discard(10)  # No error

# 4. pop()
elem = s.pop()
print("pop:", elem)   # Output: Random element, e.g., 1
print("After pop:", s)  # Remaining elements

# 5. clear()
s.clear()
print("clear:", s)  # Output: set()

# 6. copy()
s1 = {1, 2, 3}
s2 = s1.copy()
print("copy:", s2)  # Output: {1, 2, 3}

# 7. union()
```

```
a = {1, 2}
b = {2, 3, 4}
print("union:", a.union(b))  # Output: {1, 2, 3, 4}

# 8. intersection()
print("intersection:", a.intersection(b))  # Output: {2}

# 9. difference()
print("difference:", a.difference(b))  # Output: {1}

# 10. difference_update()
a.difference_update(b)
print("difference_update:", a)  # Output: {1}

# 11. symmetric_difference()
x = {1, 2, 3}
y = {2, 3, 4}
print("symmetric_difference:", x.symmetric_difference(y))  # Output: {1,
4}

# 12. symmetric_difference_update()
x.symmetric_difference_update(y)
print("symmetric_difference_update:", x)  # Output: {1, 4}

# 13. issubset()
s1 = {1, 2}
s2 = {1, 2, 3}
print("issubset:", s1.issubset(s2))  # Output: True

# 14. issuperset()
print("issuperset:", s2.issuperset(s1))  # Output: True

# 15. isdisjoint()
s3 = {5, 6}
print("isdisjoint:", s1.isdisjoint(s3))  # Output: True
```

Few Functions Used With Sets: len, sum, max, min, sorted

```
# Creating a set
numbers = {5, 1, 8, 3, 2}

# 1. len() -> number of elements in the set
print(len(numbers))  # Output: 5

# 2. sum() -> sum of all elements in the set
print(sum(numbers))  # Output: 19  (5+1+8+3+2)

# 3. max() -> maximum element in the set
print(max(numbers))  # Output: 8

# 4. min() -> minimum element in the set
print(min(numbers))  # Output: 1

# 5. sorted() -> returns a sorted list of elements
print(sorted(numbers))  # Output: [1, 2, 3, 5, 8]

# Note: sorted() returns a list, not a set
```

In Operator

In Python, the **`in` operator** is used to **check if an element exists in a set**. It returns `True` if the element is present, otherwise `False`.

```
# Creating a set
fruits = {"apple", "banana", "cherry"}

# Using 'in' to check membership
print("apple" in fruits)   # Output: True
print("orange" in fruits)  # Output: False

# Using 'not in' to check if element is NOT in the set
print("orange" not in fruits)  # Output: True
print("banana" not in fruits)  # Output: False
```

Key points:

1. `in` is **very fast** for sets because sets use **hashing**.
2. You can use `in` in **conditions**, loops, or anywhere a boolean is expected.

Frozen Sets (Immutable Sets)

- **Frozen set =** immutable set → cannot add/remove elements.

Ex :

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

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

# fs.add(4) ❌ error
```

■ Quick Summary of Collections

Data Type	Iterable	Collection	Indexable/ Sliciable	Ordered	Mutable	Uses Hashing
`str`	✅	❌	✅	✅	❌	❌
`range`	✅	❌	❌	✅	❌	❌
`list`	✅	✅	✅	✅	✅	❌
`tuple`	✅	✅	✅	✅	❌	❌
`set`	✅	✅	❌	❌	✅	✅

■ Summary of Python Collections

Property	List	Tuple	Dictionary	Set
Notation	[ ]	( )	{'Key': 'Value'}	{ }
Creation	list()	tuple()	dict()	set()
Mutability	Mutable	Immutable	Mutable	Mutable
Type of elements which can be stored	Any	Any	Keys: Hashable Values: Any	Hashable
Order of elements	Ordered	Ordered	Unordered*	Unordered
Duplicate elements	Allowed	Allowed	Keys: Not allowed Values: Allowed	Not allowed
Operations	Add, Update, Delete	None	Keys: Add, Delete Values: Add, Update, Delete	Add, Delete
Operations	Indexing, Slicing, Iteration	Indexing, Slicing, Iteration	Iteration	Iteration
Sorting	Possible	Not possible	Possible	Not possible

*Python 3.6 and earlier. Dictionaries are ordered as per Python 3.7 and above.

List methods	Tuple methods	Dictionary methods	Set methods
append() clear() copy() count() extend() index() insert() pop() remove() reverse() sort()	count() index()	clear() copy() fromkeys() get() items() keys() pop() popitem() setdefault() update() values()	add() clear() copy() difference() difference_update() discard() intersection() intersection_update() isdisjoint() issubset() issuperset() pop() remove() symmetric_difference() symmetric_difference_update() union() update()

Chapter 8 : Function , Recursion and Advance Topics in Python

Function in Python

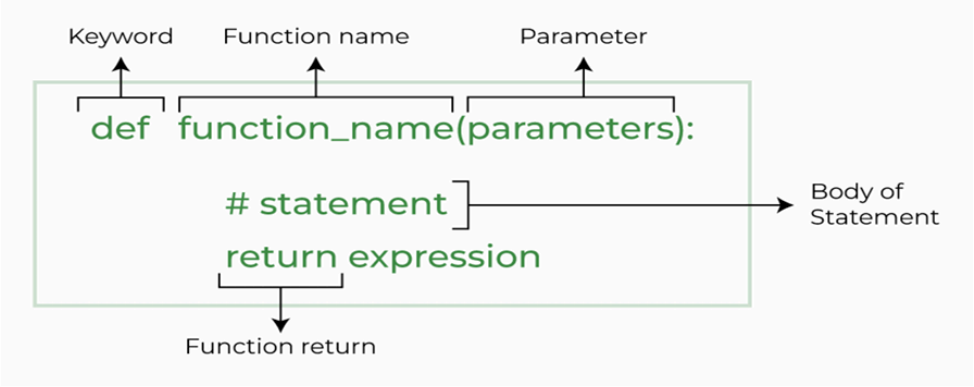
A Function is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code repeatedly for different inputs, we can do the function calls to reuse the code in it repeatedly

A **function** is a block of reusable code that performs a specific task.
Instead of writing the same code again and again, we put it in a function and call it whenever needed.

Formal definition:
A **function** is a named block of statements that can take input (parameters), perform operations, and return output.

Syntax :

```
def function_name(parameters):

    """optional docstring (explains what the function does)"""

    # block of code

    return value    # optional
```



Calling a function :

1. When the function greet() is called, the program's control transfers to the function definition.

2. All the code inside the function is executed.

3. The control of the program jumps to the next statement after the function call.

❖ **Therearethree types of functions**

- ➢ **Built-in function:** These are Standard functions in Python that are available to use. Example: `abs()`, `len()`, `max()`, `min()`, `etc.`
- ➢ **Library functions:** The functions that can be accessed by importing a library. Eg. `math.sqrt()`, `random.choice()`, `etc.`
- ➢ **User-defined function:** We can create our own functions based on our requirements.

❖ We return a value from the function using the return statement

Arguments

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

```
def greet(name):
  print("Hello,", name)
greet("Python") # Output: Hello, Python
```

Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

Ex 1 :

```
def add(x, y=100):
  return x + y
print(add(10, 10)) # Output: 20
print(add(10))     # Output: 110
```

The add function is defined with two parameters: x and y, where y has a default value of 100. When add(10, 10) is called, it explicitly passes values for both x and y, resulting in x + y which computes to 20. When add(10) is called without specifying y, the default value of y=100 is used, resulting in x + y which computes to 110.

Ex 2:

```
def greet(name="Guest"):
    return f"Hello {name}"
print(greet())          # Hello Guest
print(greet("Ayush"))   # Hello Ayush
```

Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

```
def student(name, age):
    return f"{name} is {age} years old"
print(student(age=20, name="Bruce"))
```

Positional Arguments :

We can pass the value without mentioning the parameter name but at that time the positions of the passed values matter. During the function call the first value is assigned to the first parameter and the second value is assigned to the second parameter and so on. By changing the position, or if you forget the order of the positions, function gives unexpected result.

Ex 1:

```
def sub(x, y):
  return x- y
print(sub(x=30, y=10))      # Output: 20
print(sub(y=10, x=30))      # Output: 20
print(sub(30, 10))          # Output: 20
print(sub(10, 30))          # Output: -20
```

The **sub()** function is defined to subtract **y** from **x**. Python allows function arguments to be passed by position or by keyword. In the given examples:

add(x=30, y=10) explicitly specifies the values for x and y, resulting in 30 - 10, which evaluates to 20.

add(y=10, x=30) uses keyword arguments, where the order of x and y is reversed compared to the function definition but still results in 30 - 10, also evaluating to 20 because of keyword arguments.

add(30, 10) and add(10, 30) both pass arguments by position. In the first case, the function computes 30 - 10, resulting in 20, but in the second case, the function returns 10 - 30, i.e., -20. As the argument name is not mentioned, it considers the first value as x and the second value as y.

Ex 2:

```
def power(x, y):
    return x ** y
print(power(2, 3))   # 8
print(power(3, 4))   # 27
```

Returning multiple values :

Python functions can return multiple values using one return statement. All values that should be returned are listed after the return keyword and are separated by commas.

```
def square_point(x, y, z):
 x_squared = x * x
 y_squared = y * y
 z_squared = z * z
 return x_squared, y_squared, z_squared          # Return all three values


three_squared, four_squared, five_squared = square_point(3, 4, 5)
print(three_squared, four_squared, five_squared)       # output: 9 16 2
```

Some useful built-ins:

- Math: `abs()`, `round()`, `pow()`, `sum()`, `min()`, `max()`
- Type conversion: `int()`, `float()`, `str()`, `list()`, `tuple()`, `dict()`, `set()`
- Iterables: `len()`, `sorted()`, `enumerate()`, `zip()`, `map()`, `filter()`
- Others: `print()`, `input()`, `type()`, `id()`, `help()`, `dir()`

Python Scope of Variables

1 . Python Local variable : Local variables are initialised within a function and are unique to that function. It cannot be accessed outside of the function.

```
def f():
  s = "I love Python" # local variable
  print(s)
f()
print(s)         # this line throws an error as
                 # local variable cannot be accessed outside of the
function



# Output
# I love Python
# NameError: name 's' is not defined
```

2. Python Global variable : Global variables are the ones that are defined and declared outside any function and are not specified to any function. They can be used by any part of the program.

```
def f():
  print(s)

# global scope
s = "I love Python"
f()
 # Output
 # I love Python
```

Global and Local Variables with the Same Name

If a global variable is reinitialized inside a function. It is considered as a local variable of that function and if any modification is done on that variable, the value of the global variable will remain unaffected. For example:

```
def f():
  s = "Hello World"
  print(s)

# global scope
s = "I love Python"
f()
print(s)


# Output
# Hello World
# I love Python
```

To modify the value of the global variable we need to use the global keyword

```
def f():
  global s
  s = "Hello World"
  print(s)
```

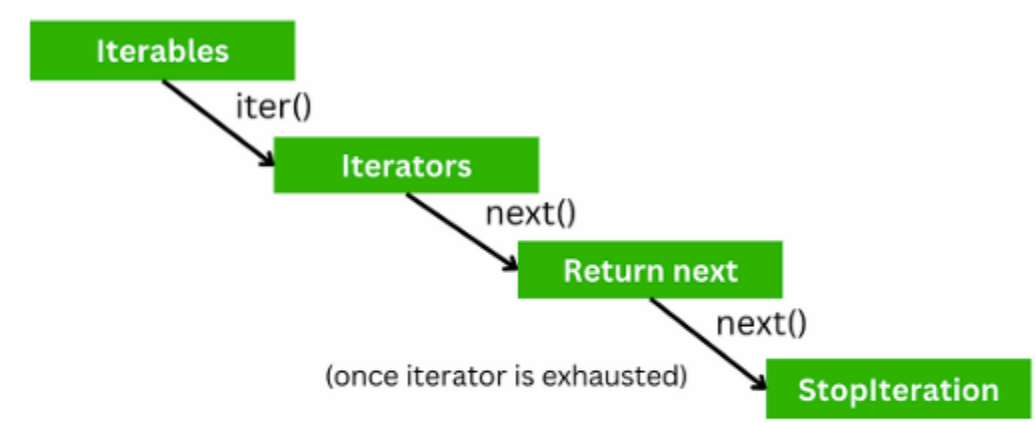
```
# global scope
s = "I love Python"
f()
print(s)



# Output
# Hello World
# Hello World
```

Iterators and Generators in Python

Iterator

In Python, an iterator is an object used to iterate over iterable objects such as lists, tuples, dictionaries, and sets. An object is called iterable if we can get an iterator from it or loop over it.

- **iter()** function is used to create an iterator containing an iterable object.
- **next()** function is used to call the next element in the iterable object.



Ex 1:

```
iter_list = iter(['I', 'Love', 'Python'])
print(next(iter_list))       # I
print(next(iter_list))       # Love
print(next(iter_list))       # Python
```

Ex 2:

```
nums = [1, 2, 3]
it = iter(nums)       # create iterator

print(next(it))   # 1
print(next(it))   # 2
print(next(it))   # 3
print(next(it))   # ❌ StopIteration
```

So, **lists, tuples, sets, dicts, strings** are all *iterables*, and you can get an *iterator* from them using `iter()`.

Note :

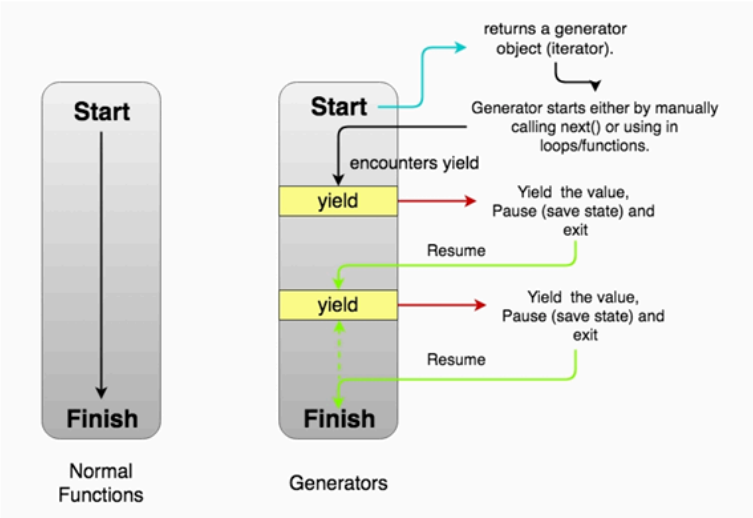
Implements **two methods**:

- `__iter__()` → returns iterator object
- `__next__()` → returns next value, raises `StopIteration` when no more items.

Generators

Python has a generator that allows you to create your iterator function. A generator is somewhat of a function that returns an iterator object with a succession of values rather than a single item. A yield statement, rather than a return statement, is used in a generator function.

The difference is that, although a return statement terminates a function completely, a yield statement pauses the function while storing all of its states and then continues from there on subsequent calls.



A **generator** is a simpler way to create an iterator using the `yield` keyword.

- Instead of returning all results at once, it yields one value at a time.
- Very **memory efficient** for large sequences.

```
# 1.
def my_gen():
    yield 1
    yield 2
    yield 3

g = my_gen()
print(next(g))   # 1
print(next(g))   # 2
print(next(g))   # 3
```

```
# 2.
def power(limit):
    x = 0
    while x<limit:
        yield x*x
        yield x*x*x
        x += 1
a = power(5)
print(next(a), next(a))         # 0 0
print(next(a), next(a))         # 1 1
print(next(a))                   # 4
print(next(a))                   # 8
print(next(a), next(a))          # 9 27
```

Python Lambda Function

Lambda Functions in Python are anonymous functions, implying they don't have a name. The def keyword is needed to create a typical function in Python, as we already know.

A **lambda function** is a **small anonymous function** in Python. It can have any number of arguments, but only **one expression**.

Syntax :

```
lambda arguments: expression
```

Example 1: Basic Lambda

```
cube = lambda x : x*x*x
print(cube(3))                      # 27



fouth_power = lambda x: x**4
print(fouth_power(3))               # 81
```

Example 2: Multiple Arguments

```
add = lambda a, b: a + b
print(add(3, 7))    # 10
```

In short :

Normal function → defined using def, reusable, can have multiple lines.

Lambda function → one-line anonymous function, often used with map(), filter(), reduce(), sorted().

enumerate() Function

The enumerate() function adds a counter to an iterable and returns it as an enumerate object (iterator with index and the value).

Syntax:

```
enumerate(iterable, start=0)
```

● iterable- a sequence, an iterator, or objects that support iteration. Eg. list, tuple, string, etc.
● start (optional)- enumerate() starts counting from this number. If start is omitted, 0 is taken as start.

Ex 1:

```
l1 = ["eat", "sleep", "repeat"]
s1 = "code"

obj1 = enumerate(l1)        # creating enumerate objects
obj2 = enumerate(s1)

print ("Return type:", type(obj1))  # Return type: <class 'enumerate'>
print (list(enumerate(l1)))    # [(0, 'eat'), (1, 'sleep'), (2,'repeat')]

# changing start index to 2 from 0
print (list(enumerate(s1, 2)))   # [(2, 'c'), (3, 'o'), (4, 'd'), (5,'e')]
```

Ex 2: Basic Use

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

for index, fruit in enumerate(fruits):
    print(index, fruit)

# output :
# 0 apple
# 1 banana
# 2 cherry
```

Ex 3: Start Index from 1

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

for index, fruit in enumerate(fruits, start=1):
    print(index, fruit)

# output :
# 1 apple
# 2 banana
# 3 cherry
```

Example 3: Convert to List

```
nums = [10, 20, 30]
print(list(enumerate(nums)))

# output :
# [(0, 10), (1, 20), (2, 30)]
```

In short :

`enumerate()` helps when you need both index and value in a loop, instead of manually using `range(len(...))`.

zip() Function

The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and returns it.

Syntax:

```
zip(iterable1, iterable2, iterable3, …..)
```

Example :

```
languages = ['Java', 'Python', 'JavaScript']
versions = [14, 3, 6]
result = zip(languages, versions)
print(list(result))
# Output: [('Java', 14), ('Python', 3), ('JavaScript', 6)]
```

- The `zip()` function combines two or more iterables (lists, tuples, strings, etc.) element-wise.
- It creates pairs (tuples) by matching items at the same index.
- Stops when the shortest iterable ends.

Example 1: Zipping Two Lists

```
names = ["Ayush", "Bruce", "Clark"]
marks = [90, 85, 88]

zipped = zip(names, marks)
print(list(zipped))

# output :
# [('Ayush', 90), ('Bruce', 85), ('Clark', 88)]
```

Example 2: Using in Loop

```
names = ["Ayush", "Bruce", "Clark"]
marks = [90, 85, 88]

for name, mark in zip(names, marks):
    print(name, "→", mark)

# output :
# Ayush → 90
# Bruce → 85
# Clark → 88
```

Example 3: Different Length Iterables

```
nums = [1, 2, 3]
letters = ['a', 'b']

print(list(zip(nums, letters)))

# output :
# [(1, 'a'), (2, 'b')]
```

Stops at the shortest iterable (`letters` has only 2 elements).

Example 4: Unzipping (Reverse of zip)

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
print(letters)  # ('a', 'b', 'c')
print(numbers)  # (1, 2, 3)
```

In short:

- `zip()` pairs elements of iterables into tuples.
- Great for looping over multiple iterables at once.
- Can be reversed using `*`.

map() Function

map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

Syntax :

```
map(function, iterables)
```

● **function-** a function that is applied to each element of an iterable.

● **iterables-** iterables such as lists, tuples, etc.

Ex 1:

```
def square(n):

    return n*n



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

```
result = map(square, numbers)

print(result)         # Output: <map object at 0x7f722da129e8>

print(set(result))    # Output: {16, 1, 4, 9}



# using lambda function in map

print(list(map(lambda x: x*x, numbers))) # Output: [1, 4, 9, 16]



num1 = [1, 2, 3]

num2 = [10, 20, 40]



# add corresponding items from the num1 and num2 lists

result = map(lambda n1, n2: n1+n2, num1, num2)

print(tuple(result))         # Output: (11, 22, 43)
```

Example 1: Using `map()` with a function

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

nums = [1, 2, 3, 4]
result = map(square, nums)
print(list(result))


# Output:
# [1, 4, 9, 16]
```

Example 2: Using `map()` with a lambda

```
nums = [1, 2, 3, 4]
result = map(lambda x: x**2, nums)
print(list(result))

# Output:
# [1, 4, 9, 16]
```

Example 3: With Multiple Iterables

If the function takes 2 arguments, you can pass 2 iterables.

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

b = [10, 20, 30]



result = map(lambda x, y: x + y, a, b)

print(list(result))

# Output:

# [11, 22, 33]
```

Example 4: Convert Strings to Uppercase

```
words = ["ayush", "bruce", "clark"]
result = map(str.upper, words)
print(list(result))

# Output:
# ['AYUSH', 'BRUCE', 'CLARK']
```

In short:

- `map()` → transforms each element of an iterable using a function.
- Useful alternative to loops when applying operations to collections.

filter() Function

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Syntax :

```
filter(function, iterables)
```

- **function**- a function that is applied to each element of an iterable.
- **iterables**- iterables such as lists, tuples, etc.

Example:

```
def check_even(number):
  return number % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# if an element passed to check_even() returns True, select it
even_numbers_iterator = filter(check_even, numbers)
print(even_numbers_iterator)      # Output: <filter object at
0x7715fbaec0d0>


# converting to list
print(list(even_numbers_iterator))       # Output: [2, 4, 6, 8, 10]

# using lambda function
print(list(filter(lambda num: num%2==0, numbers)))
# Output: [2, 4, 6, 8, 10]
```

Example 1: Filter even numbers

```
nums = [1, 2, 3, 4, 5, 6]

def is_even(x):
    return x % 2 == 0

result = filter(is_even, nums)
print(list(result))

# Output:
# [2, 4, 6]
```

Example 2: Using filter() with lambda

```
nums = [1, 2, 3, 4, 5, 6]
result = filter(lambda x: x % 2 != 0, nums)
print(list(result))

# Output:
# [1, 3, 5]
```

Example 3: Filter names with length > 4

```
names = ["bat", "superman", "flash", "ironman"]
result = filter(lambda name: len(name) > 4, names)
print(list(result))


# Output:
# ['superman', 'flash', 'ironman']
```

Example 4: Filtering truthy values

```
values = [0, 1, "", "hello", [], [1, 2]]

result = filter(None, values)   # Removes False, 0, "", [], None

print(list(result))

# Output:

# [1, 'hello', [1, 2]]
```

In short:

- `map()` → transforms each element.
- `filter()` → selects elements based on condition.

Dot Product in Python

Let two vectors (lists) are,

x = [x1, x2, x3, x4, ... , xn]
y = [y1, y2, y3, y4, ... , yn]

Dot product is defined as, $$s = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$

Eg :

```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
dot_product = 0
for i in range(len(x)):
  dot_product += x[i]*y[i]

print(dot_product)     # 70
```

Multiply Two Matrices

Let, A be a matrix of dimension R1 x C1. Here R1 represents the row of A and C1 represents the column of A. It can be denoted as A [R1 X C1].

Let A [R1 X C1] and B [R2 X C2] are matrices. The multiplication of matrices can be done if and only if C1 = R2 and the resultant matrix becomes Z [R1 X C2].

Here is the code of Matrix multiplication:

```
def matrix_multiplication(X, Y):
   # row and columns of matrix X
   r1, c1 = len(X), len(X[0])

   # row and columns of matrix Y
   r2, c2 = len(Y), len(Y[0])

   # initializing result matrix
   # the dimension of resultant matrix: r1 x c2
   result = []
   for i in range(r1):
      result.append([0]*c2)
```

```
   # matrix multiplication
   # iterate through rows of X
   for i in range(r1):
    # iterate through columns of Y
    for j in range(c2):
      # iterate through rows of Y
      for k in range(r2):
          result[i][j] += X[i][k] * Y[k][j]

    return result

# 3x3 matrix
X = [[12,7,3],[4 ,5,6],[7 ,8,9]]

# 3x4 matrix
Y = [[5,8,1,2],[6,7,3,0],[4,5,9,1]]

result = matrix_multiplication(X,Y)
for r in result:
  print(r)

# output :
# [114, 160, 60, 27]
# [74, 97, 73, 14]
# [119, 157, 112, 23]
```

Recursion

Recursion is a **programming technique** in which a function calls **itself** directly or indirectly to solve a problem.

In other words:

- A **big problem** is broken into **smaller subproblems of the same type**,
- Until a **base case** (stopping condition) is reached.

Essential Parts of Recursion

1. **Base Case** → The condition where the function stops calling itself. (Prevents infinite loop)
2. **Recursive Case** → The part where the function calls itself with a smaller/simpler input.

Process of calling a function within itself.

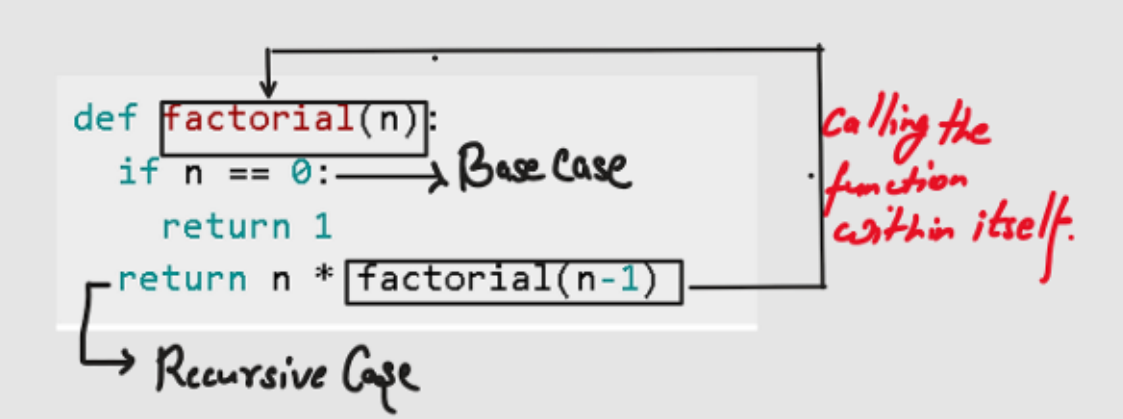
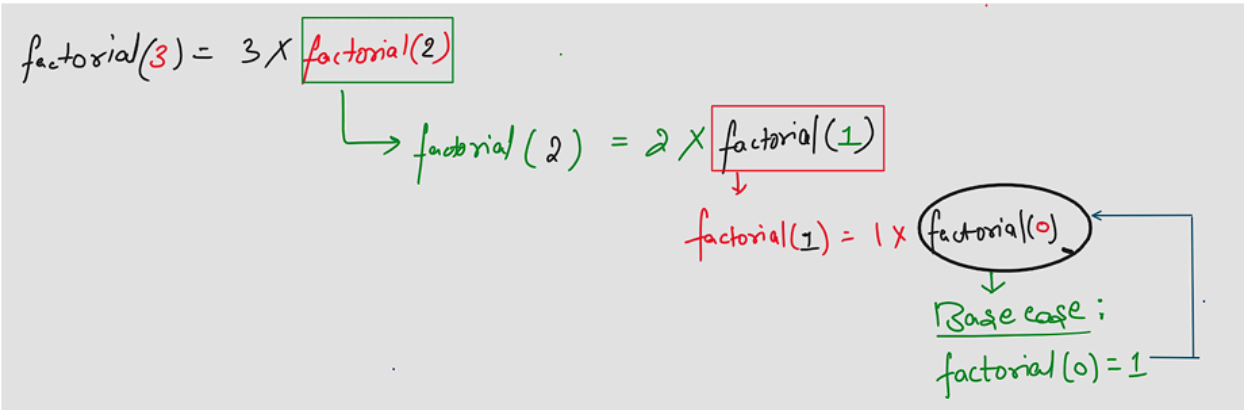


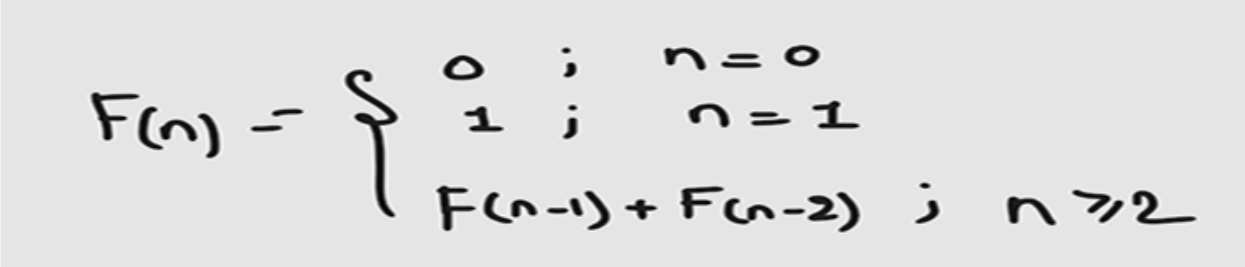
Illustration of recursion:

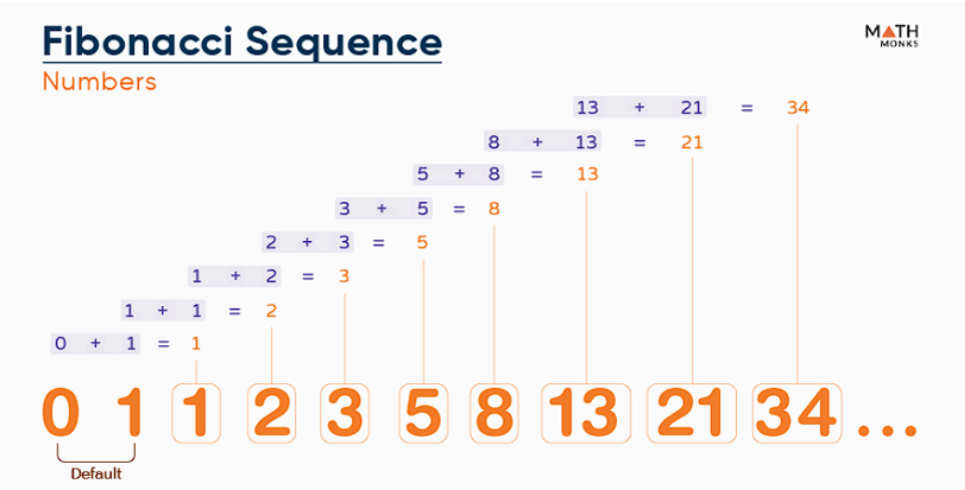



```
def factorial(n):

    if n == 0:    # base case

        return 1

    return n * factorial(n-1)
```

Fibonacci Sequence:

Series where each number is obtained by adding its two preceding numbers, starting with 0 followed by 1





Code :

```
def fibo(n):

  if n == 0: # base case

      return 0

  if n == 1: # base case

     return 1

 # recursive case

  return fibo(n-1) + fibo(n-2)
```

Values till 7

```
fibo(0) = 0

fibo(1) = 1

fibo(2) = 1

fibo(3) = 2

fibo(4) = 3
```

```
fibo(5) = 5

fibo(6) = 8

fibo(7) = 13
```

Sorting A List Using Recursion :

```
def recursive_sort(L):
  if L == [ ]: # Base Case
     return L

  # Store minimum element of list in variable `mini`.
  mini = min(L)
  L.remove(mini) # Removes `minimum element` from the list

  # Recursive Case
  return [mini] + recursive_sort(L)
```

Linear Search and Binary Search

1. Linear Search

Definition

Linear Search is the simplest searching algorithm.
It **checks every element one by one** until the desired element is found, or the list ends.

Steps

1. Start from the first element.
2. Compare each element with the target.
3. If match found → return index.
4. If end reached without match → return "not found".

Example in Python

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i    # return index
    return -1           # not found

arr = [10, 20, 30, 40, 50]
print(linear_search(arr, 30))  # Output: 2
print(linear_search(arr, 99))  # Output: -1
```

Works on unsorted or sorted lists.
Time Complexity: O(n) (worst case: check all elements).

2. Binary Search

Definition

Binary Search is a **divide-and-conquer** searching algorithm.It only works on a **sorted array**.It repeatedly divides the search interval in half.

Steps

1. Start with the middle element.
2. If target == middle → found.
3. If target < middle → search left half.
4. If target > middle → search right half.
5. Repeat until element is found or interval is empty.

Example in Python

```python
def binary_search(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2   # middle index
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1   # not found

arr = [10, 20, 30, 40, 50]
print(binary_search(arr, 30))  # Output: 2
print(binary_search(arr, 99))  # Output: -1
```

Binary Search

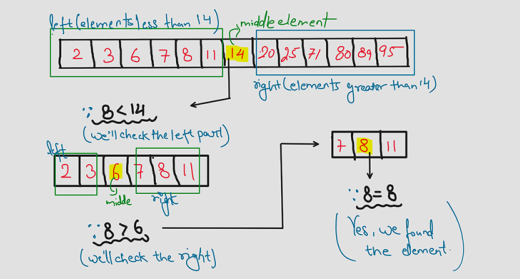
An efficient method to search the presence of an element in a sorted array.

Working:

1. Compare your target `x` with middle element.
2. If `x` == middle element we return `Element found`.
3. Else if it is **less than middle element**, then `x` can only lie in the ***left(smaller) half subarray***. So, we repeat the algorithm again in the left part.
4. Else if it is **greater than middle element**, we repeat the algorithm in the ***right part.***

   If the list becomes empty and we couldn't find the element, we return `Element not found.`

An Illustration to check presence of `8` in a sorted list:



- A real life example is when we search for a word in dictionary.
- **Note**: Binary Search is much more efficient than obvious search as we're halving the list every time.

■ An Iterative Code For Binary Search:

```
def binary_search(L, x):
    while L:  # Would repeat the process till list becomes empty.
        mid = len(L) // 2

        if L[mid] == x:
            return 'Element Found.'

        elif L[mid] > x:  # If `x` is less than middle element, then we'll
check the left half of the list.
            L = L[:mid]

        elif L[mid] < x:  # If `x` is greater than middle element, then
we'll check the right half of the list.
            L = L[mid+1:]

    # If we couldn't find the element and the list becomes empty.
    return 'Element Not Found.'
```

■ Binary Search Using Recursion :

```
def recursive_binary_search(L, x):
    mid = len(L) // 2

    if not(L):  # Base case for if element not in list (ie. list becomes
empty).
        return 'Element Not Found'

    elif L[mid] > x:    # Recursive Case (Checking the left part when `x`
< middle element)
        return recursive_binary_search(L[:mid], x)

    elif L[mid] < x:  # Recursive Case (Checking the right part when `x` >
middle element)
        return recursive_binary_search(L[mid+1:], x)

    return 'Element Found'  # Base case for element found in list.
```

Caesar Cipher

Encrypt the message

```
import string

def encrypt(s, shift):
    lower_case = list(string.ascii_lowercase)
    upper_case = list(string.ascii_uppercase)
    encrypt_msg = ''

    for char in s:
        if char in lower_case:
            encrypt_msg += lower_case[(lower_case.index(char) + shift) %
26]
        elif char in upper_case:
```

```
            encrypt_msg += upper_case[(upper_case.index(char) + shift) %
26]
        else:
            encrypt_msg += char

    return encrypt_msg

# Example usage
print(encrypt("Hello, King Caesar!!!", 3))
```

This Python code defines a function **`encrypt`** that performs a **Caesar Cipher** encryption on a given string **`'s'`** with a specified **`shift` value**. The function first creates lists of lowercase and uppercase alphabet letters. It then initializes an empty string **`encrypt_msg`** to store the encrypted message. As it iterates through each character in the input string, it checks if the character is a lowercase or uppercase letter. If so, it shifts the character by the specified **`shift`** value within the bounds of the alphabet and appends the shifted character to **`encrypt_msg`**. If the character is not a letter, it appends the character as is. Finally, it returns the encrypted message. The example given shifts each letter in "Hello, King Caesar!!!" by 3 positions, resulting in "Khoor, Nlqj Fdhvdu!!!".

Decrypt the message

```
import string

def decrypt(s, shift):
    lower_case = list(string.ascii_lowercase)
    upper_case = list(string.ascii_uppercase)
    decrypt_msg = ''

    for char in s:
        if char in lower_case:
            decrypt_msg += lower_case[(lower_case.index(char) - shift) %
26]
        elif char in upper_case:
            decrypt_msg += upper_case[(upper_case.index(char) - shift) %
26]
        else:
```

```
            decrypt_msg += char  # Keep non-alphabet characters unchanged

    return decrypt_msg

# Example usage
encrypted_text = "Khoor, Nlqj Fdhvdu!!!"
print(decrypt(encrypted_text, 3))
```

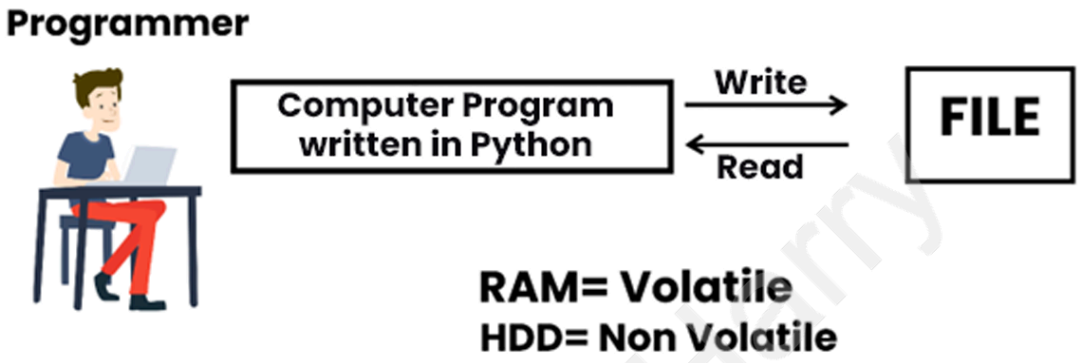
This Python code defines a function **decrypt** that performs a Caesar Cipher decryption on a given string **‘s’** with a specified **shift value.** The function creates lists of lowercase and uppercase alphabet letters and initializes an empty string **decrypt_msg** to store the decrypted message. As it iterates through each character in the input string, it checks if the character is a lowercase or uppercase letter. If so, it shifts the character backwards by the specified **shift value** within the bounds of the alphabet and appends the shifted character to **decrypt_msg.** If the character is not a letter, it appends the character as is. Finally, it returns the decrypted message. The example given shifts each letter in "Khoor, Nlqj Fdhvdu!!!!!!" backwards by 3 positions, resulting in "Hello, King Caesar!!!!!!".

Chapter 9 : File Handling in Python

File Handling in Python

The random-access memory is volatile, and all its contents are lost once a program terminates. In order to persist the data forever, we use files.

A file is data stored in a storage device. A python program can talk to the file by reading content from it and writing content to it.



TYPE OF FILES :

There are 2 types of files:

1. Text files (.txt, .c, etc)
2. Binary files (.jpg, .dat, etc)

Python has a lot of functions for reading, updating, and deleting files.

Open a File

We can open a file in various modes in Python

```
# syntax to open a file
file = open(file_path, mode)
```

Important Modes are:

Mode	Description
r	open an existing file for a read operation. If the file does not exist, it will throw an error.
w	open an existing file for a write operation. If the file already contains some data, then it will be overridden but if the file is not present then it creates the file as well.
a	open an existing file for append operation. It won't override existing data. If the file is not present then it creates the file.
r+	To read and write data into the file. This mode does not override the existing data, but you can modify the data starting from the beginning of the file.
w+	To write and read data. It overwrites the previous file if one exists, it will truncate the file to zero length or create a file if it does not exist.
a+	To append and read data from the file. It won't override existing data.

MODES OF OPENING A FILE

r – open for reading

w - open for writing

a - open for appending

+ - open for updating.

'rb' will open for read in binary mode.

'rt' will open for read in text mode.

Modes:

Mode	Meaning
"r"	Read (default). Error if file doesn't exist.
"w"	Write. Creates new file or overwrites existing.
"a"	Append. Adds data to the end of the file.
"x"	Create. Fails if file already exists.
"t"	Text mode (default).
"b"	Binary mode.
"r+"	Read and write.

Consider the ‘example.txt’ file exists with the following contents

```
Hello
How are you?
Hope you are enjoying Python.
```

Working in Read Mode

Example 1: Here we are reading a file and printing each line.

```
# Opening file in read mode
file = open('example.txt', 'r')

# Iterating line by line
for line in file:
    print(line)

# Closing the file
file.close()          # it is a good practice to close a file


# Example Output (with extra blank lines because of \n in file):
'''
Hello

How are you?

Hope you are enjoying Python.
'''
```

Here you can notice there is an extra line after each of the lines. Because in the file system, there is a **'\n'** character that gets included when we press enter to go to the new line to write

Example 2 : Here we are going to read a file and store all the lines in a list format.

```
file = open('example.txt', 'r')

# readlines() method reads all the lines of the file and
# returns in a list format where every line is an item of the list
l = file.readlines()
print(l)

file.close()

# Output
# ['Hello\n', 'How are you?\n', 'Hope you are enjoying Python.\n']
```

Here you can observe the '\n' character present at the end of the lines.

Example 3: Here we are going to read one line at a time.

```
file = open('example.txt', 'r')

# readline() method reads a line from a file and returns it as a string
first_line = file.readline()
print(first_line.strip())   # strip() removes '\n'

# Read the next line
second_line = file.readline()
print(second_line.strip())

# We can also print directly without storing it
print(file.readline())

file.close()

# Output
'''
Hello
How are you?
Hope you are enjoying Python.
'''
```

Example 4: we will extract a string that contains all characters in the Python file then we can use the read() method.

```
file = open('example.txt', 'r')
print(file.read())
file.close()

# Output

'''
Hello
How are you?
Hope you are enjoying Python
'''
```

Example 4: seek() and tell() methods.

seek() method is used to change the position of the File Handle to a given specific position. The file handle is like a cursor, which defines where the data has to be read or written in the file.

tell() method prints the current position of the File Handle.

Syntax of seek() method :

```
Syntax: f.seek(offset, from_what), where f is file pointer

Parameters:

Offset: Number of positions to move forward

from_what: It defines point of reference.

Returns: Return the new absolute position.
```

The reference point is selected by the from_what argument. It accepts three values :
0: sets the reference point at the beginning of the file
1: sets the reference point at the current file position
2: sets the reference point at the end of the file
By default from_what argument is set to 0

```
# For this case assume the 'example.txt' contains the following line:
# Code is like humor. When you have to explain it, it’s bad.

f = open("example.txt", "r")

# Second parameter is by default 0 (beginning of file).
# This moves the file pointer to the 20th index (starting from 0).
f.seek(20)

# Prints current position of the file pointer
print(f.tell())

# Reads from the 20th position until the end of line
print(f.readline())

f.close()

# Output
'''
```

```
20
When you have to explain it, it’s bad.
'''
```

Working in Write Mode

Example 1: Here we are writing a file

```
file = open('example1.txt', 'w')

file.write('new line 1')
file.write('new line 2')
file.close()

# The content of the file becomes
'''
new line 1new line 2
'''
```

write() method overrides the existing file if exists otherwise creates a new file. You can observe two lines added into a single line, if you want to add it in a different line we have to mention the **'\n'** character whenever you want to break a line.

```
file = open('example.txt2', 'w')

# Writing new content to the file
file.write('new line 1\n')
file.write('new line 2\n')

file.close()

# The content of the file becomes
'''
new line 1
new line 2
'''
```

Example 2: Write multiple strings at a time.

```
file = open('example.txt3', 'w')

# writelines() writes a list of strings into the file
file.writelines(["Item 1\n", "Item 2 ", "Item 3\n", "Item 4"])

file.close()

# The content of the file becomes
'''
Item 1
Item 2 Item 3
Item 4
'''
```

WITH STATEMENT

The best way to open and close the file automatically is the with statement .

```
# Open the file in read mode using 'with', which automatically closes the
file

with open("this.txt", "r") as f:
# Read the contents of the file
text = f.read()
# Print the contents
print(text)
```

Working in Append Mode

Example: We append a new line a the end of the file.

```
file = open('example.txt4', 'a')



# Append a new line at the end of the file
file.write('a new line will be added')



file.close()



# The content of the file becomes
'''
Hello
How are you?
Hope you are enjoying Python.
a new line will be added
'''
```

Chapter 10 : Exception Handling in Python

[Chapter 10 Google Colab]

Exception Handling in Python

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception.

When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit. In Python, we catch exceptions and handle them using try and except code blocks.

Syntax of try…..except block :

```
try:
  # code that may cause exception
except:
  # code to run when exception occurs
```

Example: Here we are trying to access the array element whose index is out of bound and handle the corresponding exception.

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

try:
  print ("Fourth element = %d" %(a[3]))
except:
  print ("An error occurred")

# Output
'''
An error occurred
'''
```

As the exception is dealt here by the **try…exception block,** the output will be shown instead of abruptly stopping the program by showing an error message.

Catching Specific Exceptions

For each try block, there can be zero or more except blocks. Multiple except blocks allow us to handle each exception differently. Please be aware that only one handler will be run at a time.

- **Generic syntax :**

```
try :
  # statement(s)
except Exception1:
  # statement(s)
except Exception2:
  # statement(s)
except:
  # if none of the above cases is matched
```

Example 1 :

```
try:
  print(5/0)

except IndexError:
  print("Index Out of Bound.")
except ZeroDivisionError:
  print("Denominator cannot be 0.")
except:
  print("Some other exception occurred")


# Output
'''
Denominator cannot be 0.
'''
```

The code attempts to divide 5 by 0, which raises a ZeroDivisionError. The try block is used to catch exceptions, and the except block handles specific errors. Since the error is a ZeroDivisionError, the corresponding except block is triggered, printing "Denominator cannot be 0." If any other exception occurs, it will be caught by the last generic except block, but in this case, it's not needed as the specific error has already been handled.

Example 2:

```
try:
  print(l)

except IndexError:
  print("Index Out of Bound.")
except ZeroDivisionError:
  print("Denominator cannot be 0.")


 # you can write the following line instead of just writing except
 # to view the error message associated with the Exception

except Exception as e:
  print(e)


# Output
'''
name 'l' is not defined
'''
```

The code tries to print the value of the variable l, which is not defined, leading to a NameError. The try block is used to handle exceptions, and the generic except Exception as e block catches the NameError, printing the error message associated with it. The other specific except blocks for IndexError and ZeroDivisionError are not triggered since they don't match the raised exception.

try block with else and finally block

● There are two optional blocks , i.e., the **else** block and **finally** block, associated with the **try** block.
● The **else** comes after the **except** block(s). Only when the try clause fails to throw an exception does the code go on to the **else** block. The **finally** block comes at the end.
● The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.

Example 1:

```
try:
  print(5//0)

except ZeroDivisionError:
  print("Can't divide by zero")
else:
  print("This line is only gets executed when there is no exception")
finally:
  print('This line is always executed')

# Output

'''
Can't divide by zero
This is always executed
'''
```

The code attempts to perform integer division by zero (5//0), which raises a ZeroDivisionError. The except block catches this error and prints "Can't divide by zero." The else block, which would execute if no exception occurred, is skipped. The finally block is then executed regardless of whether an exception was raised, printing "This line is always executed."

Example 2:

```
try:
  print(5//2)

except ZeroDivisionError:
  print("Can't divide by zero")
else:
  print("This line is only gets executed when there is no exception")
finally:
  print('This line is always executed')

# Output
'''
2
This line is only gets executed when there is no exception
This is always executed
'''
```

The code performs integer division ('5//2'), which successfully results in '2'. Since no exception is raised, the 'else' block executes, printing "This line is only gets executed when there is no exception." Finally, the 'finally' block executes, printing "This line is always executed," ensuring that it runs regardless of the outcome in the 'try' block.

Some types of exceptions in Python :

In Python, there are several built-in Python exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python :

● **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.

● **TypeError:** This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.

● **NameError:** This exception is raised when a variable or function name is not found in the current scope.

● **IndexError:** This exception is raised when an index is out of range for a list, tuple, or other sequence types.

● **KeyError:** This exception is raised when a key is not found in a dictionary.

● **ValueError:** This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.

● **AttributeError:** This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.

● **IOError:** This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error. 59

● **ZeroDivisionError:** This exception is raised when an attempt is made to divide a number by zero.

● **ImportError:** This exception is raised when an import statement fails to find or load a module.

● **FileNotFoundError:** This exception is raised when the given file is not present in the current directory

Raising Exception

The raise statement allows the programmer to force a specific exception to occur. It specifies the exception that should be raised.

Example:

```
x = 5
if x > 3:
    raise Exception("Value is greater than 3")

# Output
'''
Value is greater than 3
'''
```

The code checks if the variable x is greater than 3. Since x is 5, which satisfies the condition, an Exception is manually raised with the message "Value is greater than 3," terminating the program with this error message.

Chapter 11 : Class & Object in Python

[Chapter 11 Google Colab]

Classes and Objects in Python

Python is an **object-oriented programming language (OOP)**.
This means that everything in Python is built around the concepts of **objects** and **classes**.

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

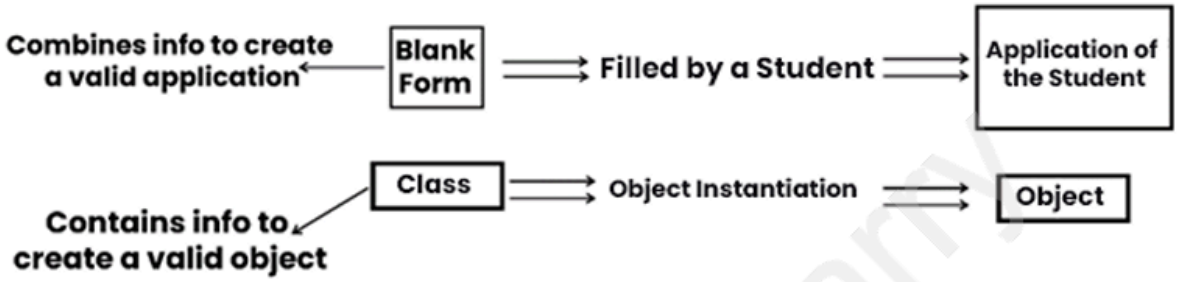
An Object is an instance of a Class. We can create multiple objects of a class

Define Python Class and Object

```
class ClassName:
  # class definition

# creating an object
obj1 = ClassName()
obj2 = ClassName()
```

CLASS

A class is a blueprint for creating object.



Syntax:

```
class Employee: # Class name is written in pascal case
    # Methods & Variables
```

OBJECT

An object is an instantiation of a class. When class is defined, a template (info) is defined. Memory is allocated only after object instantiation.

Objects of a given class can invoke the methods available to it without revealing the implementation details to the user. – Abstractions & Encapsulation!

Access attributes of Objects:

We use the **'.' (dot)** notation to access the attributes of a class.

Methods :

Objects can also contain methods. Methods in objects are functions that belong to the object.

❖ __init__() method:

This method is known as the **constructor** or **object initializer** because it defines and sets the initial values for the object's attributes. It is executed at the time of Object creation. It runs as soon as an object of a class is instantiated.

Example:

```
class Person:
    # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # methods
    def display(self):
        print(self.name, self.age)


# creating object of the class
p1 = Person("John", 36)

# accessing the attribute of the object
print(p1.name)

# using the method of the object
p1.display()


# Output
'''
John
John 36
'''
```

The code defines a Person class with a constructor **(__init__)** that initializes name and age attributes when an object is created. The class also has a display method that prints the name and age of the person. An object p1 is created with the name "John" and age 36. The name attribute of p1 is accessed and printed, resulting in "John". Then, the display method is called, which prints "John 36".

❖ self parameter :

The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class. It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.

❖ __str__() Method :

The **__str__()** function controls what should be returned when the class object is represented as a string. If the **__str__()** function is not set, the string representation of the object is returned.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Name: {self.name}, age: {self.age}"


p1 = Person("John", 36)
print(p1)

# Output
'''
Name: John, age: 36
'''
```

The code defines a 'Person' class with a constructor **('__init__')** that initializes the 'name' and 'age' attributes. It also defines a **'__str__'** method, which returns a formatted string when an instance of 'Person' is printed. When the object 'p1' is created with the name "John" and age 36, and 'print(p1)' is called, the **'__str__'** method is invoked, resulting in the output "Name: John, age: 36.”

Class and Instance Variables :

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self, whereas class variables are variables whose value is assigned in the class

Example:

```
class Student:
    # class variable
    count = 0

    def __init__(self, name, age):
        # instance variable
        self.name = name
        self.age = age
        Student.count += 1

    def __str__(self):
        return f"Name: {self.name}, age: {self.age}"


s1 = Student("John", 18)
print(s1)
print("Student count:", Student.count)

s2 = Student("Smith", 19)
s3 = Student("Alex", 17)

print(s2)
print(s3)
print("Student count:", Student.count)

# output
'''
Name: John, age: 18
Student count: 1
Name: Smith, age: 19
Name: Alex, age: 17
Student count: 3
'''
```

In the 'Student' class example, 'count' is a class variable shared by all instances of the class, used to track the total number of 'Student' objects created. Each time a new 'Student' instance is initialized, 'count' is incremented, reflecting the total number of students. In contrast, 'name' and 'age' are instance variables specific to each individual 'Student' object, storing unique attributes for each instance. While 'count' maintains a global state across all instances, 'name' and 'age' hold data specific to the object they belong to.

Methods in Class :

There are three types of methods :

1. **Instance methods** → act on object data (self).
2. **Class methods** → act on class-level data (@classmethod).
3. **Static methods** → general functions inside class (@staticmethod).

Example:

```
class Math:
    def square(self, x):  # instance method
        return x * x

    @classmethod
    def cube(cls, x):     # class method
        return x ** 3

    @staticmethod
    def add(a, b):        # static method
        return a + b

m = Math()
print(m.square(4))      # 16
print(Math.cube(3))     # 27
print(Math.add(5, 7))   # 12
```

Chapter 12 : Inheritance in Python

[Chapter 12 Colab Book]

Inheritance

Inheritance is a feature of **Object-Oriented Programming (OOP)** that allows a **class (child/subclass)** to inherit **attributes and methods** from another **class (parent/superclass)**.

Why use it?

- Reusability of code
- Reduces redundancy
- Allows extending existing functionality

● Inheritance allows you to inherit the properties of a class, i.e., parent class to another, i.e., child class.

● It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.

● It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A

General Syntax :

```
Class ParentClass:   # Base class
   {Body}
Class ChildClass(BaseClass):     # Derived or child class
   {Body}
```

Example :

```
class Person():
  def Display(self):
    print("From Person Class")


class Employee(Person):
  def Print(self):
    print("From Employee Class")
```

```
per = Person()
per.Display()

emp = Employee()
emp.Print()
emp.Display()

# output
'''
From Person Class
From Employee Class
From Person Class
'''
```

The code demonstrates inheritance in Python. The 'Person' class has a method 'Display' that prints "From Person Class." The 'Employee' class inherits from 'Person' and adds its own method 'Print' that prints "From Employee Class." An instance 'per' of 'Person' is created and calls the 'Display' method. Then, an instance 'emp' of 'Employee' is created, which calls both its own 'Print' method and the inherited 'Display' method from 'Person', demonstrating how 'Employee' can access methods from its parent class. The output reflects these method calls.

❖ Method Overriding :

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. When a method in a subclass has the same name, the same parameters or signature, and the same return type (or sub-type) as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

❖ super() function :

The super() function is a built-in function that returns the objects that represent the parent class. It allows to access the parent class's methods and attributes in the child class.

Syntax :

```
# use parent class attribute
super().attributeName

# use parent class method
super().methodName1()             # without parameter
super().methodName2(parameters)   # with parameter




## Alternate way
# use parent class attribute
ParentClassName.attributeName
# use parent class method
ParentClassName.methodName1(self)                # without parameter
ParentClassName.methodName2(self, parameters)   # with parameter
```

Example 1 :

```
class Person():
    def Display(self):
        print("From Person Class")

class Employee(Person):
    def Display(self):
        print("From Employee Class")

per = Person()
per.Display()
```

```
emp = Employee()
emp.Display()

# output
'''
From Person Class
From Employee Class
'''
```

In this example, the 'Employee' class inherits from the 'Person' class. Both classes have a 'Display' method, but the 'Employee' class overrides the 'Display' method from 'Person'. When 'per.Display()' is called on a 'Person' object, it executes the 'Display' method from 'Person', printing "From Person Class." When 'emp.Display()' is called on an 'Employee' object, it executes the overridden 'Display' method in 'Employee', printing "From Employee Class." This demonstrates that the subclass method takes precedence when called on an instance of the subclass.

Example 2 :

```
class Person():
    def Display(self):
        print("From Person Class")

class Employee(Person):
    def Display(self):
        print("From Employee Class")
        super().Display()
        # Person.Display(self)              # alternate way

per = Person()
per.Display()

emp = Employee()
emp.Display()

# output
'''
From Person Class
From Employee Class
From Person Class
'''
```

In the example, the **'Employee'** class overrides the 'Display' method of the 'Person' class with its own version. When **'emp.Display()'** is called, it first prints "From Employee Class" from the overridden method, then uses **'super().Display()'** to call the 'Display' method from the 'Person' class, which prints "From Person Class." This allows 'Employee' to extend the functionality of 'Person' while still using the parent class's method.

❖ __init__() function :

When the **__init__()** function is added in the child class, the child class will no longer inherit the parent's **__init__()** function. To keep the inheritance of the parent's **__init__()** function, add a call to the parent's **__init__()** function.

Example :

```
class Person():
    def __init__(self, name, id):
        self.name = name
        self.id = id

    def Display(self):
        print("Name:", self.name, "ID:", self.id)

class Employee(Person):
    def __init__(self, name, id, salary):
        super().__init__(name, id)
        self.salary = salary

    def Display(self):
        print("Name:", self.name, "ID:", self.id, "Salary:", self.salary)

emp = Employee("Dan", 123, 10000)
emp.Display()

emp.salary = 15000
emp.Display()

# output
'''
Name: Dan ID: 123 Salary: 10000
Name: Dan ID: 123 Salary: 15000
'''
```

The code demonstrates inheritance and method overriding, with 'Employee' as a subclass of 'Person'. The 'Person' class has an '__init__' method to initialize 'name' and 'id', and a 'Display' method to print them. The 'Employee' class overrides the '__init__' method to include 'salary' and uses 'super().__init__(name, id)' to call the parent class's constructor. It also overrides the 'Display' method to print 'name', 'id', and 'salary'. When an 'Employee' object 'emp' is created and its 'Display' method is called, it prints the initial salary. After updating 'emp.salary', calling 'Display' again reflects the updated salary in the output.

❖ Private Variables in Python:

In definition, private variables would be those that can only be seen and accessed by members of the class to which they belong, not by members of any other class. When the programme runs, these variables are utilized to access the values to keep the information secret from other classes. Even the object of that class cannot access this private variable directly. It is only accessible via any method. In Python, it is accomplished using two underscores before the name of the attributes or the method.

Example 1 :

```
class Person():
    def __init__(self, name, id):
        self.name = name
        self.__id = id   # making the variable private

    def Display(self):
        print("Name:", self.name, "ID:", self.__id)

    def changeID(self, id):
        self.__id = id

# creating a object and show details
p = Person('Nobita', 1)
p.Display()

# changing the value of id directly
p.__id = 2
# it will not throw any error but nothing will be changed
# as private attributes are not directly accessible even if by own object
```

```
p.Display()

# if you want to change any private variables, you have to use
# some method that can access that private variables.
# Because private variables are accessible inside any method
# of that class, but not any child/sub class.
p.changeID(3)
p.Display()

# the following line throws an error as private methods are accessible by
# methods inside the class but not accessible by object.
# p.__privateMethod()

# output
'''
Name: Nobita ID: 1
Name: Nobita ID: 1
Name: Nobita ID: 3
'''
```

Example 1:

```
class Person():
    def __init__(self, name, id):
        self.name = name
        self.__id = id   # making the variable private

    def Display(self):
        print("Name:", self.name, "ID:", self.__id)

    def showID(self):
        return self.__id

class Employee(Person):
    def __init__(self, name, id, salary):
        super().__init__(name, id)
        self.salary = salary
```

```
    def Display(self):
        print("Name:", self.name)
        print("Salary:", self.salary)
        # following line throws an error as private variable is not
accessible
        # print("ID:", self.__id)
        print("ID:", self.showID())  # access via parent method

        # parent class method can be accessed by 'self.methodName()'
        # if no other method is present in the child class with same name

emp = Employee("Dan", 123, 10000)
emp.Display()

# output
'''
Name: Dan
Salary: 10000
ID: 123
'''
```

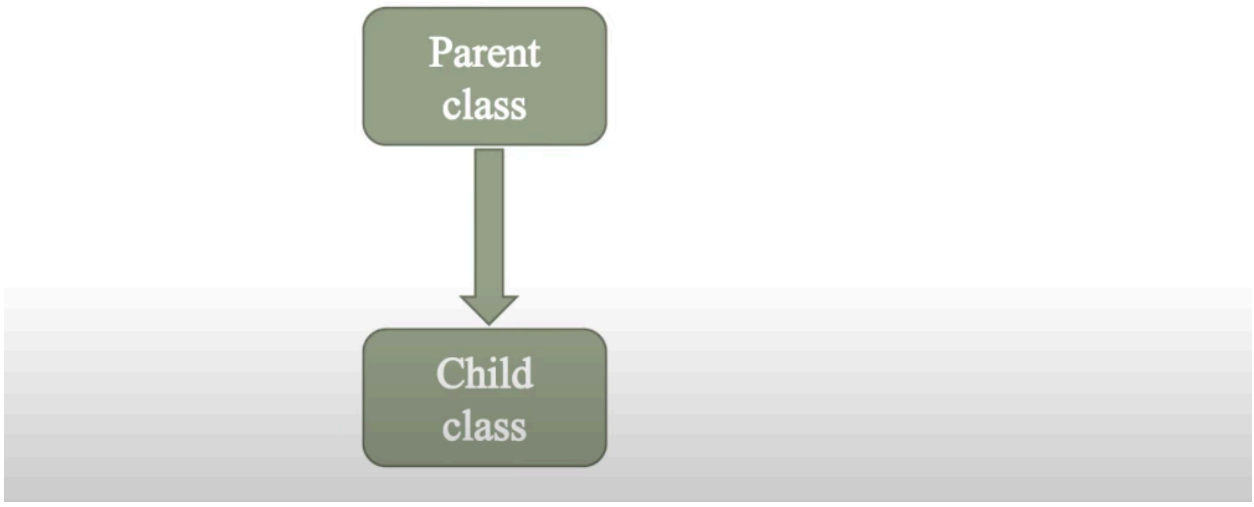
Types of Inheritance :

1. Single Inheritance:

A child class inherits from a single parent class.

Syntax:

```
class Parent:
   pass
class Child(Parent):
   pass
```

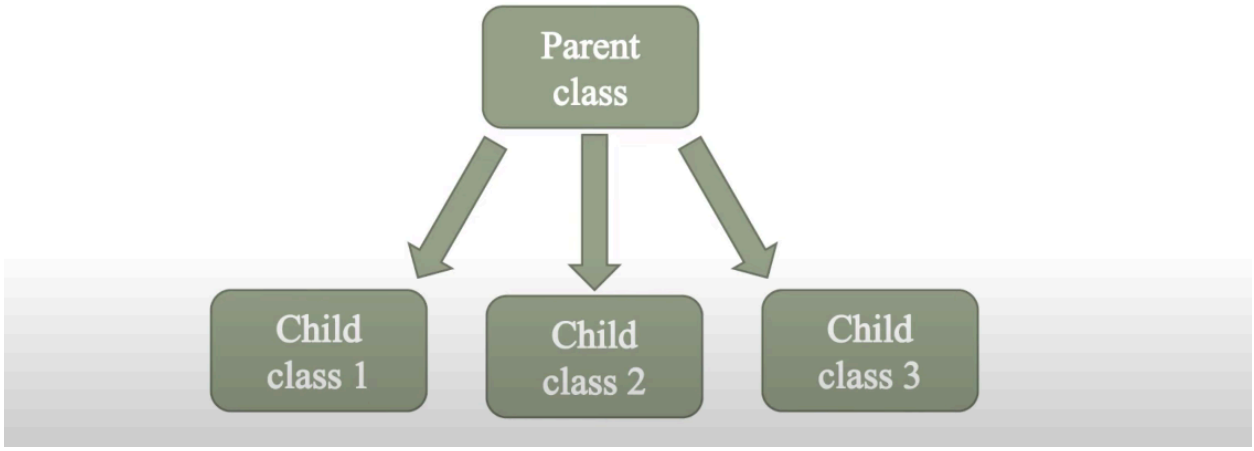


2. Hierarchical Inheritance:

Multiple child classes inherit from the same parent class.

Syntax:

```
class Parent:
   pass

class Child1(Parent):
   pass

class Child2(Parent):
   pass
```

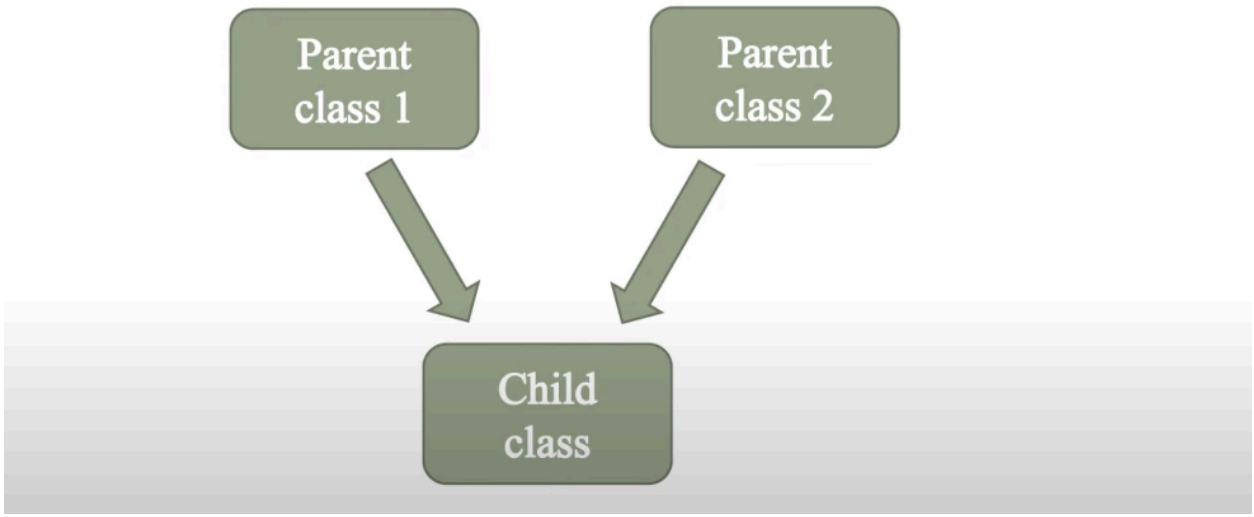


3. Multiple Inheritance:

A child class inherits from more than one parent class.

Syntax :

```
class Parent1:
   pass

class Parent2:
   pass

class Child(Parent1, Parent2):
   pass
```

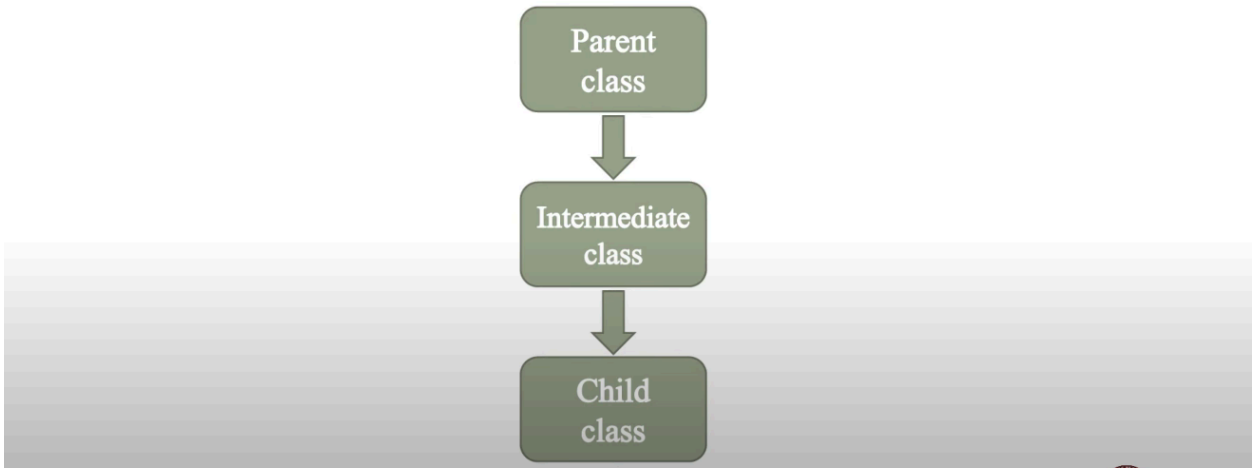


4. Multilevel Inheritance:

A child class inherits from a parent class, which in turn inherits from another parent class.

Syntax:

```
class Grandparent:
   pass

class Parent(Grandparent):
   pass

class Child(Parent):
   pass
```



5. Hybrid Inheritance:

A combination of two or more types of inheritance.

Example:

```
class Parent1:
   pass

class Parent2:
   pass

class Child(Parent1, Parent2):
   pass

class GrandChild1(Child):
   pass

class GrandChild2(Child):
   pass
```