PYTHON PROGRAMMING

Prepared by

ABDUL MUQSITH

CHAPTER - 1 PYTHON INTRODUCTION

1.8 Introduction to Python Programming Language

Python is a very popular and easy-to-learn programming language. It was created by **Guido van Rossum** in 1991 in Netherlands .It is derived from many other languages including ABC,C,C++,Algol,UNIX shell. Python is a **high-level** and **interpreted** language, which means the computer reads and executes the code line by line. Its main advantage is that it has **simple English-like syntax**, so beginners can learn it very easily. Python is used in many fields of software development and engineering

---

🎯 Why Should We Learn Python?

Python is a powerful and flexible language. It is **free and open-source**.
It has a **large library** of built-in functions and third-party modules. With Python, we can do:

- Web development (using Flask or Django)
- Data analysis and visualization
- Artificial Intelligence and Machine Learning
- Games (Vega strike) and GUI applications
- Automation of daily tasks
- Language development (swift)

In short, Python is a "one language, many uses" type of programming language.

Python 3.8	**2019**	**Added walrus operator (:=) — lets you write shorter and smarter code.**
Python 3.9	2020	Made it easier to combine two dictionaries using `
Python 3.10	2021	Introduced **match-case** (like switch-case in C), and better error messages.

Python vs Java

No.	Feature	Python	Java
1	**Syntax**	Simple and clean; uses indentation instead of {}	Uses braces {}, semicolons ;, and is more verbose
2	**Ease of Learning**	Beginner-friendly; easy to write and understand	Slightly complex; more rules to follow

3	**Code Length**	Short and readable; fewer lines of code	Longer programs for the same task
4	**Typing System**	Dynamically typed (no need to declare variable type)	Statically typed (must declare data types)
5	**Speed of Execution**	Slower due to interpretation line by line	Faster because it is compiled to bytecode
6	**Best Use Cases**	Data Science, AI, Automation, Scripting, Education	Android apps, Web applications, Enterprise software

7	**EXAMPLE**	print("hello")	Public class A {
			public static void main(string[] Args){
			System.out.println("hello worlds");
			}
			}

1.11 Features of Python.

1. **Easy to learn and use :** Python is the beginner friendly language. It is easy to learn and use. Its statements are more similar to English language.
2. **Easy to understand : The** python code is easy to understand as its statements are simpler
3. **Easy to debug :** in case of error , it is easy to find the error and the debug the error.
4. **Free and open source: Python is** freely available on its official website so no need of any 3rd party sources to install it, just a programmer can download it from its official site https://www.python.org
5. **Portable**: python is a portable language (platform independent) which means the python code written in one platform like windows , the same code can be executed in to the another platform like mac OS etc, which makes python the portable language
6. **Python is dynamically typed**: no need to declare variable type while writing the source code, during runtime variable type is specified by the compiler itself
7. **Various applications:** used in various applications like AIML , automation, software development, GUI designing, Web designing (Django); etc;
8. **Object oriented:** python supports classes and object and othe characteitics of OOPs so its an OOPL

(EEEPOPFV)

WhO USE PYTHON TODAY: Google,Youtube,Raspberry Pi,Intel,Cisco,Qualcomm,IBM,Nasa

Chapter-2 Basic of Python Programming

2.1 Declare and Initialize Variables

Variable: A **variable** is a **container** that **stores data** in memory.
The data can be of any type: ➡️ Integer, Float, String, Boolean, etc.

Python is Dynamically Typed language so

- you **don't need to declare the type** of the variable.
- The **data type is automatically assigned at run-time** based on the value you store.

✅ This is called a **dynamically-typed language**.

Rules for Python Variables declaration (Identifiers):

1. Must start with a letter (A–Z or a–z) or an underscore (_)
2. Cannot start with a digit (0–9)
3. Can contain only letters, digits, and underscores (A–Z, a–z, 0–9, _)
4. Cannot contain spaces or special characters (such as @, #, $, %, &, etc.)
5. Cannot use Python reserved keywords as variable names(such as if, for, class, def, True, etc.)

6. Variable names are case-sensitive

```
Syntax:  variable_name = value
```

Note: For storing string or char in a variable use "xyz" double quotes ex name="caliph"

`examples`

Name="zaid bin usman" →**string**

Age =20 →**integer**

Earnings=100000.2938 →**float**

What is a Comment?

- A **comment** is a line that is **not executed** by Python.
- It is used to **explain the code** so makes long and complex code understandable
- Comments **help in understanding** the logic, especially when code becomes complex.

✅ Single-line Comment:

Use # at the beginning of the line. Used when a comment length is about one line

```
# This is a comment

x = 10  # This is also a comment
```

✅ Multi-line Comments:

used if comment extends from more than 1 line

```
"""

This is a multi-line

comment-like string

often used for documentation

"""
```

Note: The triple-quoted string is still technically a string, not a true comment — but if it's not assigned or used, Python ignores it.

What is Indentation?

- **Indentation is the whitespace (space) space at the beginning of a line of code**.
- In Python, **indentation is compulsory** and used to **define blocks** like loops, functions, if-statements, etc.
- In a block each line of code must have the same identation(space) , uncommon space represent end of block
- Other languages use { } to group code, but **Python uses indentation**.

✅ Example of Correct Indentation:

$if\ True$:

    $print("Hello")$ $\#\ indented\ inside\ the\ if-block$

    $print("World")$

❌ Example of Incorrect Indentation:

$if\ True$:

$print("Hello")$ $\#\ This\ will\ give\ an\ IndentationError$

✅ **Common Indentation Rules:**

- Default: **4 spaces** (recommended)
- All lines in the same block **must have same indentation**

✅ **Example:**

$x = 5$

$if\ x > 0:$

$print("Positive")$

$print("Number")$ $\#\ Same\ block$

$print("Done")$ $\#\ Outside\ the\ if\ block$

2.3 DATA TYPES

Data type = the type of the data that a variable stores in the memory is called data type

In Python they are

1. Numbers
2. Strings
3. Lists
4. Dictionaries
5. Tuples
6. Sets
7. Booleans
8. None

1. NUMBERS: python has 3 built in numeric datatypes:

   - Integers – these represents that the variable is like {-1,2,0,922,-18,12093 etc}
   - Float - these data types represents that the variable is the decimal values like 1.2, 101.8939 etc
   - Complex- these data types represents that the variable is complex number that's in the form x+iy where x is the real part and the iy is imaginary part.

❖ Example :

```
a=10
b=10.5
c=3+9j
print(“integer is ”a)
print(“float is ”b)
print(“complex  is ”c)
```

2. SEQUENCE type: data types contains ordered collection of elements

❖ Strings : strings is the data type that represents that the variable is storing the group of characters

- The variable value must be in double quotation marks
- Example :

```
name=”Muqsith”
print(“my name is:”,name)
```

❖ Lists: Lists are **ordered collections** of items. These items can be of **same or different types** such as numbers, strings, or even other lists.

- **Declaration**: Lists are declared using **square brackets []**.
- **Mutability**: Lists are **mutable**, meaning their elements can be changed after creation.
- Examples :

```
fruits = ["apple", "banana", 100,2000]
                print(fruits)
                print(fruits[0])
```

Outputs: ['apple', 'banana', 'cherry']
banana

❖ **Tuples: Definition**:

A **tuple** is an **ordered collection** of items, similar to a list, but **immutable**. This means once a tuple is created, its elements **cannot be changed**.

- **Declaration**: Tuples are declared using **parentheses ()**.

- **Immutability**: Tuples are **immutable**, meaning you **cannot add, change, or remove** elements after creation.

```
fruits = ("apple", "banana", "cherry")
print(fruits)
print(fruits[1])
```

  - OUTPUTS : ('apple', 'banana', 'cherry')
    banana

- NOTE:

```
single = ("apple",)  # This is a tuple
not_a_tuple = ("apple")  # This is just a string in parentheses
```

3. **Dictionaries** : A **dictionary** is an **unordered collection** of **key-value pairs**.
   Each item has a **key** and a **value**, like a word and its meaning.

   - **Declaration**:
     Dictionaries are declared using **curly braces {}**, with **key: value** pairs.
   - **Mutability**:
     Dictionaries are **mutable**, meaning you can **add, update, or delete** items.

```
student = {"name": "bhai", "age": 21, "course": "Python"}
print(student)
print(student["name"])

Output: {'name': 'bhai', 'age': 21, 'course': 'Python'}
        bhai
```

4. Sets: A **set** is an **unordered collection** of **unique items**.
   It means **duplicates are not allowed** in sets.

   - **Declaration**:
     Sets are declared using **curly braces {}** or the **set() constructor**.
   - **Mutability**:

     Mutable sets : Sets are **mutable**, which means you can **add or remove elements**, but the elements themselves must be **immutable** (e.g., numbers, strings, tuples).

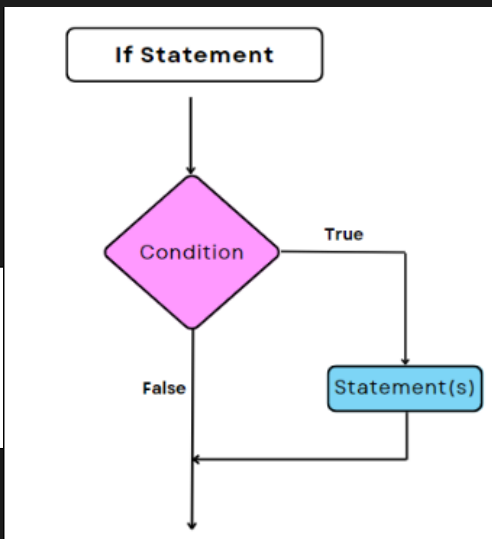
     In Python, you cannot put mutable objects inside a set, because sets require all elements to be hashable (fixed) — and mutable objects are not hashable. Also, since sets are **unordered**, they do not support indexing or slicing. Means we cant access single element

```
numbers = {1, 2, 2, 3, 3, 3}
print(numbers)
```

Output: {1, 2, 3}

Example 2
my_set = {[1, 2, 3]} # ❌ Error: list is mutable

OUTPUT: TypeError: unhashable type: 'list'



5. Boolean Type

bool – Only two values: True or False

```
a= 5
b = 3

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

Output:

True
False
```

6. NONE:

Decision making statements : decide **which block of code should be executed** based on given **certain conditions**.

It gives your program **a control flow ;**

Python supports:

1. If statement
2. If-else statement
3. Nested if statements
4. Multiway if-else statement.

1. IF STATEMENT: it states that if some condition s satisfied then execute the the code written in if block or else skip the if block

Syntax:

```
If condition:

        Statements(s)
```

➔ The block of code inside 'if ' must maintain IDENTATION (common space before beginning of line) the first uncommon space from the beginning of line represents end of block.

```
age = 20

if age >= 18:

   print("You are eligible to vote.")

OUTPUT: You are eligible to vote.
```

2. IF...ELSE STATEMENT:

It states that if a given condition is **satisfied**, then execute the code written in the **if block**, **otherwise**, execute the code written in the **else block**.

It helps handle **two-way decisions**:

```
➔ age = 16

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

    else:
        print("You are not eligible to vote.")

▶ OUTPUT: You are not eligible to vote.
```

⚠ **Note on Indentation:**
In Python, indentation is used **instead of brackets** to define blocks.
All lines inside if or else must be **uniformly indented**.

3) NESTED IF...ELSE STATEMENT:

```python
age = 20
citizen = True

if age >= 18:
    if citizen:
        print("You are eligible to vote.")
    else:
        print("You must be a citizen to vote.")
else:
    print("You are too young to vote.")
```

Output: you are not eligible to vote

4. IF…ELIF…ELSE STATEMENT (Multi-way decision):

The **if…elif…else** statement is used when we want to **check multiple conditions one by one**.
It executes the block of the **first true condition** and **skips the rest**.
If **no condition is true**, the else block is executed

```
marks = 72

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

⚠️ Important Points:

- ➔ Only the first true condition is executed.
- ➔ You can have multiple elif blocks, but only one if and one else.
- ➔ All blocks must be properly indented.

2.5 loop statements wih examples

Looping statements: A looping statement is used to execute a block of code repeatedly as long as a given condition is true. It helps reduce code repetition and improves efficiency.

Types of Looping Statements in Python:

1. for loop
2. while loop
3. Nested loops

while Loop

A while loop runs **as long as a given condition is true**. It is used when the number of iterationns is **not known** beforehand.
suntax:

```
initialization
while:
    code
    modification statement
```

```
i = 1
while i <= 3:
    print("Welcome")
    i += 1
```

Output:

```
Welcome
Welcome
Welcome
```

For Loop:

The block of code inside a for loop is executed **multiple times based on the number of items in the sequence or the range. In short used when we know the number of iterations**

- A for loop is like "repeat this for every item in the list or range."

Is for loop based on condition?

No – A for loop in Python does not depend on a condition like a while loop does.

Syntax:

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

The *for* loop automatically picks each item from the *sequence* (it can be a number, list of items, tuples string) and stores it in *variable* temporarily for that round. Then executes the block for that round until *the sequence or range* reaches end.

In- Another keyword. It connects the variable and the sequence.

Example:

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

Output:

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

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

for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

```
numbers = (10, 20, 30)

for num in numbers:
    print(num)
```

Output:

```
10
20
30
```

Explanation: for fig 1

- The loop goes through the **string letter by letter**.
- Each time, char holds one character from "Python".
- And it execute the block of code

Explanation: for fig 2

- The loop goes through the **list of items**.
- fruit takes each item one by one.
- And it execute the block of code

Explanation: for fig 3

- Just like a list, but here we are looping through a **tuple** (round brackets).
- num gets each value from the tuple one by one.

RANGE: IN Python, range() is a built-in function used to generate a sequence of numbers. It **starts from 0 by default**, **increases by 1** each time (by default), and **stops before** the specified stopping value.

range(start,stop,step) is **built-in** function with 3 parameters

Step in range = "Step defines how much to increase or decrease the parameter $start\ of\ range$ and loop variable after each iteration of the loop"

- If you give only one number like range(5), it means:
  start = 0, stop = 5, step = 1. (increment start by 1)

- But if you give 3 numbers like range(1,10,2) it means:
  start = 1, stop = 10, step = 2. (increment $start$ by 2 means after each iteration the $start$ incremented by 2)

- The loop will **not include** the stopping number.
- It is mainly used in for loops
- And it execute the block of code

python

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

Here:

- **start** = 1
- **stop** = 10 (not included)
- **step** = 2

Nested loops;

If the loop statement is **written inside the another loop** then its called **as nested loop**

```
Ex:
for I in range (1,6):
    for j in range(1,i+1):
        print(j end="")
    print()
```

★ Pattern Printing Using Single Loop in Python

☑ Main Idea

Instead of using nested loops, we use:

string **number**. **Print("hello"***4) → hello\n hello\n hello\n hello\n

1 Square Pattern (Single Loop)

Pattern	Code

* * * *
* * * *
* * * *
* * * *
	`n = 4`
	
	`for i in range(n):`
	`  print("* " * n)`

2 Left Triangle Pattern (Single Loop)

Pattern	Code	Logic

*
* *
* * *
* * * *
	`n = 4`
	
	`for i in range(1, n+1):`
	`  print("* " * i)`
		• Stars increase
		• Stars = i

3 Reverse Triangle (Single Loop)

Pattern	Code	Logic
* * * * * * * * * *	**n = 4** **for i in range(n, 0, -1):** ** print("* " * i)**	• Stars decrease • Stars = i

4 Right Triangle (Single Loop)

Pattern	Code	Logic
* * * * * * * * * *	**n = 4** **for i in range(1, n+1):** ** print(" " * (n-i) + "* " * i)**	• Spaces = n - i • Stars = i

Important Formulas Summary

Pattern Type	Formula
Square	`"* " * n`
Increasing	`"* " * i`
Decreasing	`"* " * i` (reverse loop)
Right Triangle	`" " * (n-i) + "* " * i`

2.6 Perform Operations On List, Tuples,Sets And Dictionaries

Lists: Lists are **ordered collections** of items. These items can be of **same or different types** such as numbers, strings, or even other lists.

```
Syntax:   list_name = ["listmember1", listmember2, listmember3]
```

NOTE: declare string or char under straight double quotes " or single quotes '

Explanation:

List elements can be **strings, numbers, characters, or even other lists**. They are similar to arrays in C, but Python lists are **more flexible**, as they can hold elements of **different types** in the same list.

Example:

```
fruits = ["apple", "banana", 100,2000]
                print(fruits)
                print(fruits[0])
```

Index: order[location] of list

Operation	Syntax	What It Does (including negative behavior)	What It Returns	Works with
Create List	list = [1,2,3]	Creates a new list	New list object	
Access Element	list[index] ex: list[-1]→ 3	Gets element at given index. Negative index (-1, -2) accesses from end	Element value	List dictionary tuple
Slicing	list[start:end] ex: list[-2:-1]→ [2,3]	Gets part of list. Negative values allowed (-1, -3)	New sliced list	List tuple
Modify Element	list[index] = value ex: list[2]=”hi”	Changes element at index (negative index allowed)	Nothing (modifies list)	List dictinary
Append	list.append(x)	Adds element at end	None	list
Insert	list.insert(i, x)	Adds element at position (negative index allowed)	None	list
Extend	list.extend(iterable)	Adds multiple elements at end	None	list
Remove	list.remove(x)	Removes first occurrence of x	None	List , set
Pop	list.pop()	Removes last element (same as pop(-1))	Removed element	list

Pop with Index	list.pop(i)	Removes element at index. Negative index removes from end	Removed element	list
Delete	del list[i] out of bound is ignored	Deletes element or slice (negative index allowed)	Nothing	List, dictinary
Clear	list.clear()	Removes all elements	None	List set dictionary
Index	list.index(x) list.index(“hi”)	Returns first position of x	Integer index	List,tuple
Count	list.count(x)	Counts occurrences of x	Integer	List,tuple
Sort	list.sort()	Sorts list ascending (modifies original)	None	list
Sort Descending	list.sort(reverse=True)	Sorts descending (modifies original)	None	list
Reverse	list.reverse()	Reverses order (modifies original)	None	list
Copy	list.copy()	Creates shallow copy	New list	List, dictinary set
Join (Concatenation)	list1 + list2	Merges two lists	New combined list	List tuple
Membership Test	x in list	Checks if element exists	True / False	All for dictinary check keys
List Comprehension	Syntax: expp for item in iterable	Creates new list using expression	New list	All return list
Length	len(list)	Counts total elements	Integer	All
Nested Access	list[i][j]	Access inner element (negative allowed at both levels)	Inner element	All except set

Example:

```
# 10. Joining (Concatenation)

list1 = [1, 2]

list2 = [3, 4]

joined = list1 + list2

print("Joined list:", joined)
```

```
# 11. Membership Test
print("Is 100 in my_list?", 100 in my_list)   # True or False


# 12. Length of list
print("Length of my_list:", len(my_list))     # Number of elements


# 13. List Comprehension
squares = [x**2 for x in range(5)]
print("Squares using list comprehension:", squares)


# 14. Nested Lists
nested = [[1, 2], [3, 4]]
print("Nested element [1][0]:", nested[1][0])  # Access 2D element (3)
```

Tuples

Operation	Method / Syntax	Short Purpose
Repetition	tuple * n also for list	*Repeat tuple/list elements n times*
Nested Access	tuple[i][j]	*Access item in nested tuples*
Unpacking	a, b = tuple	*Assign elements to multiple variables*
Conversion to List	list(tuple)	*Convert tuple to list for editing*
Using in Loop	for item in tuple:	*Iterate through each element*
Immutability	❌ (no direct method)	*Cannot modify or delete elements*

```
# ==============================
#     Python Tuple Operations
# ==============================


# 1. Creating a tuple
my_tuple = (10, 20, 30, "hello", 3.14)
```

```
print("Original Tuple:", my_tuple)          # Output: (10, 20, 30, 'hello',
3.14)



# 2. Accessing elements

print("First element:", my_tuple[0])        # Output: 10

print("Last element:", my_tuple[-1])        # Output: 3.14



# 3. Slicing

print("Sliced tuple [1:4]:", my_tuple[1:4]) # Output: (20, 30, 'hello')



# 4. Length of tuple

print("Length of tuple:", len(my_tuple))   # Output: 5



# 5. Count occurrences

print("Count of 20:", my_tuple.count(20))  # Output: 1



# 6. Index of element

print("Index of 'hello':", my_tuple.index("hello"))  # Output: 3



# 7. Membership test

print("Is 30 in tuple?", 30 in my_tuple)   # Output: True

print("Is 100 in tuple?", 100 in my_tuple) # Output: False



# 8. Concatenation

tuple1 = (1, 2)

tuple2 = (3, 4)
```

```
joined = tuple1 + tuple2
```

```
print("Concatenated tuple:", joined)       # Output: (1, 2, 3, 4)



# 9. Repetition

repeated = tuple1 * 3

print("Repeated tuple:", repeated)         # Output: (1, 2, 1, 2, 1, 2)



# 10. Nested tuple access

nested_tuple = ((10, 20), (30, 40))

print("Nested element:", nested_tuple[1][0])  # Output: 30 here 1 is outer tuple
index 0 is inner tupple index



# 11. Tuple unpacking

x, y = (5, 15)

print("Unpacked values:", x, y)            # Output: 5 15



# 12. Convert to list

converted_list = list(my_tuple)

print("Tuple converted to list:", converted_list)

# Output: [10, 20, 30, 'hello', 3.14]



# 13. Looping through a tuple

print("Looping through my_tuple:")

for item in my_tuple:

    print(item)

# Output:

# 10

# 20

# 30
```

```
# hello

# 3.14



# 14. Immutability test (will cause error if uncommented)

# my_tuple[0] = 99  # ❌ TypeError: 'tuple' object does not support item assignment
```

SET:

Define two sets

1. A = {1, 2, 3, 4}
2. B = {3, 4, 5, 6}

1. Union

print(A | B) # {1, 2, 3, 4, 5, 6} → all elements from both sets

2. Intersection

print(A & B) # {3, 4} → common elements

3. Difference

print(A - B) # {1, 2} → in A but not in B

print(B - A) # {5, 6} → in B but not in A

4. Symmetric Difference

Print(A ^ B) # {1, 2, 5, 6} → in A or B but not in both

5. Subset

print(A <= B) # False → A is not a subset of B

6. Proper Subset

print({1, 2} < A) # True → {1, 2} is a proper subset of A

7. Superset print(A >= {1, 2}) # True → A contains {1, 2}

✅ **Important Dictionary Operations – Quick Table**

Operation	Method / Syntax	Short Purpose
Create Dictionary	**{key: value, ...}**	*Define a dictionary with key-value pairs*
Access Value	**dict[key]**	*Get value by key*

Add/Update Value	**dict[key] = value**	*Insert new or update existing key-value pair*
Delete Key	**del dict[key]**	*Remove key and its value*
Get with Default	**dict.get(key, default)**	*Safely get value; avoid KeyError*
Keys Only	**dict.keys()**	*Get all keys*
Values Only	**dict.values()**	*Get all values*
Items (Pairs)	**dict.items()**	*Get all key-value pairs*
Check Membership	**key in dict**	*Check if key exists*
Loop Through Dictionary	**for k, v in dict.items():**	*Iterate over key-value pairs*

```python
# 1. Creating a dictionary
student = {"name": "Alice", "age": 20, "grade": "A"}
print("Original dictionary:", student)
# Output: {'name': 'Alice', 'age': 20, 'grade': 'A'}

# 2. Accessing a value
print("Student name:", student["name"])  # Output: Alice

# 3. Adding a new key-value pair
student["city"] = "Delhi"
print("After adding city:", student)
# Output includes 'city': 'Delhi'

# 4. Updating an existing value
student["grade"] = "A+"
print("After updating grade:", student)
# Output: 'grade': 'A+'

# 5. Deleting a key
del student["age"]
print("After deleting age:", student)
# Output no longer includes 'age'

# 6. Getting a value safely using get()
print("City:", student.get("city", "Not Found"))       # Output: Delhi
print("Phone:", student.get("phone", "Not Found"))     # Output: Not Found

# 7. Getting all keys
print("Keys:", list(student.keys()))    # Output: ['name', 'grade', 'city']

# 8. Getting all values
print("Values:", list(student.values()))  # Output: ['Alice', 'A+', 'Delhi']
```

Copy Edit

```python
# 9. Getting all items (key-value pairs)
print("Items:", list(student.items()))
# Output: [('name', 'Alice'), ('grade', 'A+'), ('city', 'Delhi')]

# 10. Membership test
print("Is 'name' a key?", 'name' in student)   # Output: True
print("Is 'age' a key?", 'age' in student)     # Output: False

# 11. Looping through dictionary
print("Looping through dictionary:")
for key, value in student.items():
    print(key, ":", value)
# Output:
# name : Alice
# grade : A+
# city : Delhi
```

Question 13 (Tricky One 👀)

Code:

```
tuple = {}

tuple[1,2,4] = 8
tuple[(4,2,1)] = 10
tuple[(1,2)] = 12
```

🔑 Very Important:

```
tuple = {}
```

This is NOT tuple ❌ This is a dictionary ✓

Because `{}` creates dictionary.
So now we are adding keys:
Dictionary becomes:

```
{
  (1,2,4): 8,
  (4,2,1): 10,
  (1,2): 12
}
```

Now if they run:

```
for k in tuple:
    sum += tuple[k]
```

It will add values: 8 + 10 + 12 = 30

Question 12

```
sampleList = [10, 20, 30, 40]
del sampleList[0:6]
print(sampleList)
```

🔑 Important Idea:

List has only 4 elements.

del sampleList[0:6] means:

Python slicing does NOT give error if range is larger. It deletes everything available.

☑ Final o/p: [ ]

🔥 1 List Reference Trap

```
a = [1, 2, 3]
b = a
b[0] = 10
print(a)
```

Question: What is output? ***Ans: [10,2,3]***
in python objects assignment doesnot create new object but it acts as a pointer pointing to same memory to which it has assigned
so changes made by b also reflects to a

🔥 2 List Copy Trick

```
a = [1, 2, 3]
b = a[:]
b[0] = 10
print(a)
```

✅ **Output:**

[1, 2, 3]

✓ Justification:

a[:] creates **new list (shallow copy)**.
Modification in b does NOT affect a.

❓ Does normal copy cause referencing?

Yes.
b = a → referencing
b = a[:] → new object

🔥 3 Nested List Trap

```
a = [[0]*3]*3

a[0][0] = 5

print(a)
```

✅ **Output:**

[[5, 0, 0], [5, 0, 0], [5, 0, 0]]

✓ Justification:

*3 repeats same inner list reference.
All rows point to same object

🔥 4 Tuple Confusion

```
t = (1, 2, 3)

print(t[1:2])
```

✅ **Output:** (2,)

✓ Justification:

Slicing always returns same type.
Tuple slicing → tuple.

❓ For list?

a = [1,2,3] print(a[1:2])

Output: [2]

🔥 5 Single Element Trap

```
t = (5)
print(type(t))
```

✅ **Output:** <class 'int'>

✓ Justification:
No comma → not tuple.
Correct single tuple → (5,)

🔥 6 Set Duplicate Logic

```
s = {1, 2, 2, 3, 3, 3}
print(len(s))
```

✅ **Output:** 3 **✓ Justification:**Set removes duplicates automatically.

🔥 7 Empty Set Trap

```
s = {}
print(type(s))
```

✅ **Output:** <class 'dict'>

✓ Justification:
{} creates dictionary.
Empty set → set()

🔥 8 Tuple Multiplication

t = (1, 2)
print(t * 2)

✅ **Output:** (1, 2, 1, 2)

✓ **Justification:** Multiplication repeats tuple elements.

🔥 9 List + Tuple Trick

a = [1, 2]
b = (3, 4)
print(a + list(b))

✅ **Output:**[1, 2, 3, 4]

✓ **Justification:**

Tuple converted to list → valid concatenation.
Without list(b) → ❌ TypeError.

🔥 10 Set Order Confusion

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

✅ **Answer:** Order NOT guaranteed.

✓ **Justification:**
Set is unordered collection.

ECET

MEMBERSHIP OPERATORS: In Python, the **membership operators** in and not in are used to check whether a value exists in a **list**, **tuple**, **set**, or **dictionary**.

✅ **1. List**

```
my_list = [1, 2, 3, 4]

print(2 in my_list)     # True

print(5 not in my_list)  # True
```

✅ **2. Tuple**

```
my_tuple = (10, 20, 30)

print(20 in my_tuple)      # True

print(25 not in my_tuple)   # True
```

✅ 3. Set

```
my_set = {100, 200, 300}

print(200 in my_set)     # True

print(400 not in my_set)  # True
```

✅ 4. Dictionary

```
my_dict = {'a': 1, 'b': 2}

print('a' in my_dict)        # True (checks for key)

print(2 in my_dict)          # False (values not checked by default)

print(2 in my_dict.values())  # True (explicit value check
```

also u can use if to print like

```
my_dic={'a':"ahmed", 'b':25}

if 'a' in my_dic:

    print("a is in the dictionary i.e",my_dic['a'])
```

Useful Built-in String Functions

Function	Description	Example
len(s)	Length of string	len("hello") → 5
s.lower()	Convert to lowercase	"HELLO".lower() → hello
s.upper()	Convert to uppercase	"hello".upper() → HELLO
s.strip()	Remove whitespace from both ends	" hi ".strip() → "hi"
s.replace(a,b)	Replace a with b	"hi yaaro".replace("hi", "hello")
s.find(sub)	Find position of substring	"apple".find("p") → 1
s.split()	Split into list (by space or given char)	"a b c".split() → ['a','b','c']
s.isalpha()	Checks if all characters are alphabetic	"abc".isalpha() → True
s.isdigit()	Checks if all characters are digits	"123".isdigit() → True

```
# Sample string operator

name = "  YaAro  "

# Using Operators
```

```
greeting = "Hi " + name         # + operator CONCATENATION
repeat = "Hello! " * 2          # * operator MULTIPLY 2 TIMES
is_in = "a" in name             # in operator  SEARCH A
equals = name.strip() == "YaAro" # == operator

# string[start:end:step] [2:9:2]

text = "Yaaro"

print(text[0])      # 'Y'

print(text[1:4])    # 'aar' (1 TO 3 INDEX 4 IS EXCLUDED)

word = "Python"

# Normal iteration

for char in word:

   print(char)

# Iteration with slicing (like manual loop)

for i in range(len(word)):

   print("Index", i, "has:", word[i]
```

Output:

Greeting: Hi YaAro

Repeated: Hello! Hello!

'a' in name: True

Trim equals 'YaAro': True

Char at 0 : L

Char at 2 : a

Char at 4 : n

Char at 6 : P

Char at 8 : h

Char at 10 : n

2.9 Build functions with /without arguements

In Python, **functions** are reusable blocks of code that perform a specific related tasks. You can define them **with or without arguments**.

- When a function is written inside any class its called method and a method is called using a object or class

General Syntax for Function Definition:

```
def function_name(parameters):

   # block of code (function body)

   # statements

   ...
```

🔍 **Explanation:**

- def: Keyword used to define a function
- function_name: Name you choose for your function (should follow naming rules)
- parameters: Optional. Variables listed inside parentheses to receive input (can be empty)
- :: Colon indicates the start of the function body
- NO NEED TO DECLARE FUNCTION

Function calling : using syntax → **functionname(parameters)** you can call the function where ever you need its operation in your program

Function without arguments

A **function without arguments** does **not take any input parameters when** it is defined or called.

🧱 **Syntax:**

```
def function_name():

   # code block

   ...
```

💡 **Example:**

```
def greet():
   print("Hello, welcome to Python!")



greet()  # Calling the function
```

Function with arguments

🧱 **Syntax:**

```
def function_name(arg1, arg2, ...):
    # code block
    ...
```

💡 **Example:**

Function with multiple arguments

```
def add(a, b):
    result = a + b
    print("Sum is:", result)
add(5, 7)
```

a function can return multiple values using return val1,val2,val3

Pass by Value:

- A **copy** of the variable is passed.
- Changes **inside the function do not affect** the original variable.
- Common in languages like **C (for basic types)**.

Pass by Reference in Python

In **pass by reference**, a function **receives the reference (memory address)** of the variable, not just its value.
So if the object is **mutable**, changes made inside the function **affect the original variable** outside the function.

In Python, this happens with **mutable objects** like:

- list
- dict
- set
- custom classes.

🧪 **Simple Example: Pass by Reference using list**

```
def update_list(my_list):
    my_list.append(99)
    print("Inside function:", my_list)
```

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

update_list(numbers)

print("Outside function:", numbers)
```

🟢 **Output:**

```
Inside function: [1, 2, 3, 99]

Outside function: [1, 2, 3, 99]
```

Chapter - 3 Classes and Packages

Functions and OOPs

- **Functions** and **OOPs** were introduced to improve coding by **reducing redundancy** and **increasing reusability**.
- **OOPs** (Object-Oriented Programming System) was designed to **map real-world scenarios** into code using **objects**.

Basic OOP Concepts

Class

- A **class** is a **blueprint or prototype** of an object.
- It defines the **structure**, **attributes**, and **methods** an object will have.
- **Syntax**
  Syntax for class definition:

```
#creating class
        class Student:
            age=18
#object
s1=Student()
print(s1.age)
```

Object

- An object is an instance (real version) of a class.
- It gives real values to the properties defined in the class.

Constructor: it is a function that executes by default when object is created

In python, while creating Classes we define _ _ *init*_ _ named Function with self as parameter
This __init__() which is always executed when any Object is being initiated ,

- It always takes minimum one parameter called self parameter,
- **self** is a parameter that represents the **current object** of the class.
- It **links the current object** with the class's **variables, attributes, and methods**, so that their values get updated **for that particular object** when it is created with passing parameters.
- Self is also used to access variables belongs to that class ,
- Without self, the variable will be treated as a **local variable** inside the method, not linked with the object.

Example:

```
 # definition of class
class Student:
    College_name = "GIOE"                # class Variable (common for all objects)
    name = "Not_given"                    # default attribute (used if name not provided)


    def __init__(self, name):                    # constructor method
        self.name = name                # instance attribute (unique per object)


    def show(self):                                # instance method
        print("Hi, I am", self.name)


# Create object or instance
s1 = Student("Ahmed")
s2 = Student("Ali")
s1.show()    # Output: Hi, I am Ahmed
s2.show()    # Output: Hi, I am Ali
```

- Example: if we write `s1.show()`, Python internally runs it as `Class.show(s1)`. Here, `s1` is passed as `self`.
- Later, if we call `s2.show()`, then `s2` becomes `self`.
- So, `self.variable_name` means that the variable value is different for different objects

Type	Description	Example
Class Variable	**Same for all objects**	**College_name**
Instance Variable	**Unique for each object**	**self.name**

Best Example for defining class with attributes and methods and creating its instances
Marks of students as list and name

```
#class definition
class Student_memo:
    def __init__(self, name, marks):
        print("You can enetr name and marks as a list")
        self.name=name
        self.marks=marks
    def show(self):
        print("printing the name and marks...")
        print(self.name)
        i=0
        for sub in self.marks:
            i =i+1
            print(f"subject {[i]}:",sub)

#object creation
s1=Student_memo("Abdul",[100,99,99])
s2=Student_memo("Adnan",[98,96,89])
s3=Student_memo("Ajmal",[99,98,97])
s3.name=”Rafay” #name of ajmal changed to rafay
s1.show()
s2.show()
s3.show()
```

4 Pillars of OOPs

1. **Abstraction** – Hiding internal details and showing only the main features.

```
class Car:
    def __init__(self):
        self.clutch = False
        print("Car not started")
    def start(self):
        self.clutch = True
        print("Car started")
car1 = Car()
car1.start()   # Output: Car started
```

2. **Encapsulation** – Wrapping **data + functions** together into a single unit.
    - Example: Bank Account (balance + deposit + withdraw)
3. **Inheritance** – Deriving properties from another class.
4. **Polymorphism** – Using one function name in different ways depending on the object or data type.

Private Members

- To make an attribute or method **private**, prefix it with __.
- These are **not accessible directly** from outside the class.

Example:

```
class Person:
    __name = "anonymous"        # Private attribute
    def __hello(self):          # Private method
        print("Hello Person")


    def greet(self):            # Public method
        self.__hello()          # Accessing private method


p1 = Person()


p1.greet()              # ☑ Allowed → Output: Hello Person
# print(p1.__hello())   ❌ Not allowed
# print(p1.__name)      ❌ Not allowed
```

Private members can only be accessed **inside the class**, not directly by the user.

✅ Question 11

```
class tester:
    def __init__(self, id):
        self.id = str(id)
        id = "224"

temp = tester(12)
print(temp.id)
```

Self.id ≠ id

so as we are passing 12 it is treated as objects variable as 12 .
Isliye jab hum temp.id i.e.., obj.id print karte, toh current object ka id print hota.
Lekin agar hum tester.id print kare toh error aata, kyunki id constructor ke andar hai. Woh sirf local variable rehta, class variable nahi hota.
Agar id class variable hota, toh phir class ke name se bahar se bhi access kar sakte the. Like print(tester.id)

```
class A:
   def __init__(self, x):
      self.x = x
      x = x + 5

obj = A(10)
print(obj.x)
```

as this x is 10 as its we are accessing the objects var

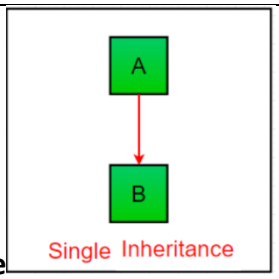
```
class A:
   def __init__(self, x):
      self.x = x
x =5

obj = A(10)
print(A.x)     //5
```

3.2 Inheritance

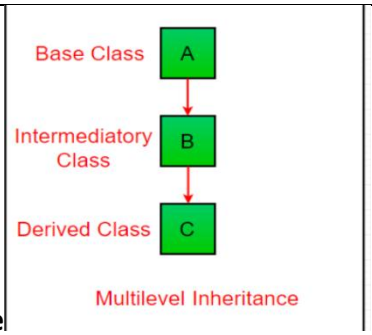
Definition:
Inheritance allows one class to **use the properties and methods** of another class.

- **Parent/Base Class:** Class whose properties are inherited.
- **Child/Derived Class:** Class that inherits the properties.

Type	Description

Single Inheritance 	One derived class inherits from one base class.
	`class Base:`
	`    statements`
	
	`class Derived(Base):`
	`    statements`

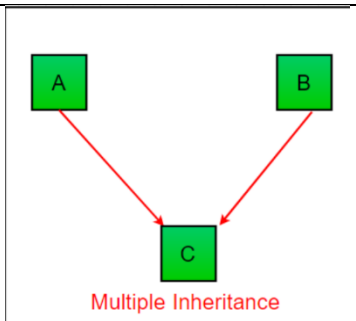
Multi-Level Inheritance



A class is derived from another derived class.

```
class Base:
    statements

class Intermediate(Base):
    statements

class Derived(Intermediate):
    statements
```

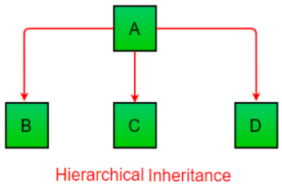
Multiple Inheritance



A class inherits from more than one base class.

```
class Base1:
    statements

class Base2:
    statements

class Derived(Base1, Base2):
    statements
```

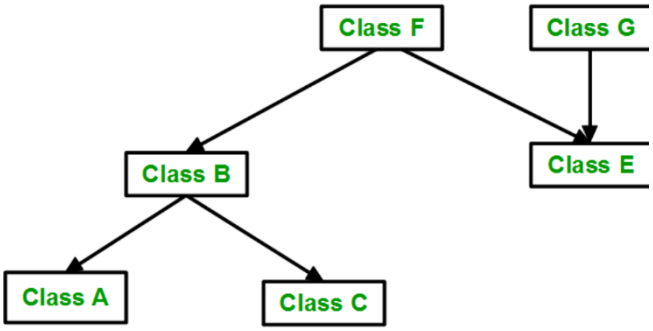
Hierarchical Inheritance



Multiple derived classes inherit from a single base class.

```
class Base:
    statements

class Derived1(Base):
    statements

class Derived2(Base):
    statements
```

Hybrid Inheritance



Combination of two or more inheritance types.

```
class Base:
    statements

class Derived1(Base):
    statements

class Derived2(Base):
    statements

class Derived3(Derived1, Derived2):
    statements
```

SINGLE INHERITANCE

```
class Car:
    def start(self):
            print("Car started...")
    def stop(self):
        print("Car stopped...")
class Toyota(Car):
    def __init__(self, color):
        self.color = color
        print("Toyota car created with color:", self.color)
# Object creation
t1 = Toyota("Red")
# Calling methods
t1.start()
t1.stop()
```

Hierarchical inheritance

```
class Car:
    def start(self):
          print("Car started...")
    def stop(self):
         print("Car stopped...")
class Toyota(Car):
    def __init__(self, color):
        self.color = color
        print("Toyota car created with color:",
self.color)
class Honda(Car):
    def __init__(self, color):
        self.color = color
        print("Honda car created with color:",
self.color)
t1 = Toyota("Red") # Object creation
h1 = Honda("Blue")
t1.start()
t1.stop()
h1.start()
h1.stop()
```

Multi level inheritance

```
class Vehicle:
    def fuel_type(self):
        print("This vehicle runs on petrol or diesel.")


class Car(Vehicle):
    def start(self):
        print("Car started...")
    def stop(self):
        print("Car stopped...")


class Toyota(Car):
    def __init__(self, color):
        self.color = color
        print("Toyota car created with color:", self.color)
# Object creation
t1 = Toyota("White")


# Using methods from all levels
t1.fuel_type()   # From Vehicle (grandparent)
t1.start()      # From Car (parent)
t1.stop()       # From Car (parent)
```

Multiple

```
class Car:


    def start(self):
          print("Car started...")
    def stop(self):
          print("Car stopped...")
class Engine:
    def engine_info(self):
          print("This engine is a hybrid engine.")
class Toyota(Car, Engine):
    def __init__(self, color):
          self.color = color
          print("Toyota car created with color:",
self.color)
# Object creation
t1 = Toyota("Black")
# Using methods from both parent classes
t1.start()        # From Car
t1.engine_info()   # From Engine
t1.stop()         # From Car
```

3.3 Use super to call methods of a super class

SUPER() : super() is a **built-in Python function** that gives a **child class** access to its **parent class methods and constructor so they can assign values and access values**.

```
class Parent:
```

```
    def __init__(self, p_name):
        self.p_name = p_name   # Use the correct parameter
        print(f"Parent initialized with name: {self.p_name}")


class Child(Parent):

    def __init__(self, p_name, c_name, age):
        super().__init__(p_name)   # Call Parent's constructor
        self.c_name = c_name
        self.age = age
        print(f"Child initialized with name: {self.c_name} and age:
{self.age}")
# Object creation

c1 = Child("Abdulla", "Muhammad", 25)
```

3.4 python identity operator

Definition:

Identity operators in Python are used to **compare the memory locations** of two objects — basically to check if **both variables refer to the same object** in memory.

Operators:

Operator	Description
is	Returns **True** if both variables point to the **same object**
is not	Returns **True** if both variables point to **different objects**

a = [1, 2, 3]
b = a # b refers to same list object as a
c = [1, 2, 3]

ex: 1

print(a is b) # True → both point to same object
print(a is c) # False → same value but different objects as in python these are mutable
print(a is not c) # True → different memory locations

ex 2:

car1 = "Toyota"
car2 = "Toyota"
car3 = car1
print(car1 is car2) # True (because strings are immutable & Python reuses memory)
print(car1 is car3) # True (both point to same object)

3.6 Use Local and Global Variables

1. Global Variables

- **Definition:** Variables defined **outside of all functions or blocks**.
- **Scope:** Accessible **anywhere in the program**, inside or outside functions.
- Modifying a global variable inside a function → will NOT affect the original global variable **unless you use the global** keyword**.**

```
x = 10   # global variable
def show():
    print("Inside function, x =", x)
```

```
show()        # Output: Inside function, x = 10
print(x)      # Output: 10
```

2. Local Variables

- **Definition:** Variables defined **inside a function**.
- **Scope:** Only accessible **inside that function** where they are defined.
- **Usage:** Use for temporary calculations or values specific to that function.

Example:

```
def loop():
       i=0
       for i in range(5):
                  print(i)          #inside function local variable
loop() # prints 0 1 2 3 4
print(i) #Outside Function ❌not accessible
```

3.5 Create and Import Modules and Packages

- ✅ One class
- ✅ One variable
- ✅ One function

📦 Step 1: Create a Module File

Create a file named:

mymodule.py

Inside it, write this:

```
# 🔹 Variable
company_name = "OpenAI Learning"

# 🔹 Function
def greet(name):
    return f"Hello {name}"

# 🔹 Class
class Student:
    def __init__(self, name):
        self.name = name

    def show(self):
        return f"Student Name: {self.name}"
```

📦 Step 2: Use the Module in Another File

Create another file:

main.py

Write this:

```
import mymodule

# Access variable
print(mymodule.company_name)

# Access function
print(mymodule.greet("Ali"))

# Access class
obj = mymodule.Student("Rahul")
print(obj.show())
```

🖨 Output

```
OpenAI Learning
Hello Ali
Student Name: Rahul
```

File & System Handling

os	Interact with operating system
sys	Access system-level info

Math * : for maths operation and calculations.

- *import math*

 # 🔢 Number & Rounding

```
print("Ceil:", math.ceil(4.2))       # Ceil: 5
print("Floor:", math.floor(4.8))     # Floor: 4
print(math.floor(-4.2))  # -5
print("Trunc:", math.trunc(4.9))     # Trunc: 4
print("Absolute:", math.fabs(-7.5))   # Absolute: 7.5
```

 # 🧮 Power & Log

```
print("Power:", math.pow(2, 3))      # Power: 8.0
print("Square Root:", math.sqrt(16))  # Square Root: 4.0
print("Log (base e):", math.log(10))  # Log (base e): 2.302585092...
print("Log10:", math.log10(100))     # Log10: 2.0
print("Exponential:", math.exp(2))    # Exponential: 7.389056...
```

 # 📐 Trigonometry (Angles converted to radians)

```
print("Sin 90°:", math.sin(math.radians(90)))  # Sin 90°: 1.0

print("Cos 0°:", math.cos(math.radians(0)))    # Cos 0°: 1.0
```

```
    print("Tan 45°:", math.tan(math.radians(45)))  # Tan 45°: 0.999999...
    asin, acos, atan exist for inverse

# 🔁 Angle Conversion

    print("Radians of 180°:", math.radians(180))   # Radians of 180°: 3.141592...
    print("Degrees of π:", math.degrees(math.pi))  # Degrees of π: 180.0

# 🧩 Special Functions

    print("Factorial:", math.factorial(5))  # Factorial: 120
    print("GCD:", math.gcd(12, 18))        # GCD: 6
    print("LCM:", math.lcm(12, 18))        # LCM: 36
    print("Integer sqrt:", math.isqrt(17))  # Integer sqrt: 4

# 📌 Constants

    print("Value of pi:", math.pi)    # Value of pi: 3.141592...
    print("Value of e:", math.e)      # Value of e: 2.718281...
    print("Infinity:", math.inf)      # Infinity: inf special class which returns x on x/0
    print("Not a number:", math.nan)  # Not a number: nan
```

Function Type	Negative Allowed?
Rounding	✅ Yes
Trigonometry	✅ Yes
sqrt()	❌ No
log()	❌ No
factorial()	❌ No

3.10 Use datetime Package in Python Application

◇ No	Concept	Code	Example Output
1	Get Current Date & Time	import datetime now = datetime.datetime.now() print(now)	2025-10-04 17:25:30.123456
2	Get Only Date	import datetime today = datetime.date.today() print(today)	2025-10-04
3	Get Only Time	import datet imecurrent_time = datetime.datetime.now().time() print(current_time)	17:25:30.123456

4	Create Specific Date	import datetime my_birthday = datetime.date(2003, 8, 15) print(my_birthday)	2003-08-15
5	Format Date & Time	import datetime now = datetime.datetime.now() print(now.strftime("%d-%m-%Y %H:%M:%S"))	04-10-2025 17:25:30

Chapter -4 Exception handling and Multithreading

4.1 Difference between compile time errors, runtime errors and logical errors

Feature	Compile Time Error	Runtime Error	Logical Error
When it Occurs	Before execution (during code checking where syntax is verified by the interpreter/compiler before conversion to machine code)	During program execution (when the code with correct syntax performs an invalid operation)	After execution (output stage — no syntax or runtime issue, but logical mistake in code)
Cause	Syntax mistakes or wrong code structure	Invalid operations like division by zero, file not found, etc.	Wrong logic or incorrect formula
Example	print("Hello" → missing bracket	10 / 0 → division by zero	area = l + b instead of l * b
Program run/not	Program doesn't run	Program stops while running	Program runs but gives wrong output
Detection	Detected by the Python interpreter before running	Detected when that part of code executes	Hard to detect — only seen from incorrect output
Also called as	Syntax Error, parsing	Execution error	semantic error

4.2 List common compile time errors and runtime errors

These happen **before the program runs**, mostly due to syntax mistakes or wrong code structure:

1. **SyntaxError** → Typing mistakes in code structure

```
print("Hello"  # missing closing bracket
```

2. **IndentationError**
3. **NameError** → Using a variable or function that's not defined
4. **TypeError** → Incorrect data type operation detected early

```
print("Hello" + 5)  # cannot add string and integer
```

5. **ImportError / ModuleNotFoundError** → Importing a module that doesn't exist EX: maths

---

Common Runtime Errors in Python

These happen **while the program is running** due to invalid operations:

1. **ZeroDivisionError** → Dividing by 0
2. **FileNotFoundError** → Trying to access a file that doesn't exist ex: open("data.txt")
3. **IndexError** → Accessing a list/array index that doesn't exist

```
ex: lst = [1, 2, 3]
print(lst[5])
```

4. **KeyError** → Accessing a dictionary key that doesn't exist

```
d = {"a": 1}
print(d["b"])
```

5. **ValueError** → Wrong type of value in a function

```
Ex: int("hello")  # cannot convert string to int
```

4.3 Using try/except, finally and else block to handle exceptions

🛑 Python Exception Handling — Blocks Overview

🔹 No	Block	Key Information

1	`try`	Contains code that may raise an exception. Executes first. No other blocks without try
2	`except`	Handles the exception if it occurs in `try`. Prevents program crash.
3	`else`	Executes only if no exception occurs in `try`.
4	`finally`	Always executes, whether exception occurs or not. Used for cleanup.

🎯 **Flow Summary**

Situation	Execution Order
Exception occurs	`try → except → finally`
No exception	`try → else → finally`

Full Example (All Blocks Together)

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful:", x)
finally:
    print("This will always execute")
```

ex:2

```
try:

file = open("data.txt")          #open file
data = file.read()

except FileNotFoundError:
print("File not found!")       #if error occurs then avoid crash

finally:

print("This will always execute")
file.close()
```

4.4 Usage of raise statement

The raise statement is used to **manually trigger an exception** in Python.
It's useful when you want to **force an error** if something goes wrong in your program.
Syntax

raise ExceptionType("Error message")

- ExceptionType → the type of exception you want to raise (e.g., ValueError, TypeError or just Exception)

- "Error message" → optional message describing the error
- It doesnot require try blocks

Example 1: Raising a ValueError

```
age = -5
if age < 0:
    raise ValueError("Age cannot be negative!")
Output:
ValueError: Age cannot be negative!
```

Example 2: Raising a Custom Exception

```
def check_score(score):
    if score > 100:
        raise Exception("Score cannot be greater than 100")

check_score(150)
Output:
Exception: Score cannot be greater than 100
```

4.5 Create User defined exception classes

Meaning:

- In Python, **User Defined Exceptions** are used to create custom errors created by the programmer.
- In this method we use class where we inherit built-in Exception class so, when we raise define exception and raise exception , and if the exception met then the raise statement stops execution and control is given to except block

◇ **Syntax:**
```
class MyException(Exception):
    pass
```

Example 1: Basic Custom Exception

```
class AgeError(Exception):
   def __init__(self, message):   # constructor to store message
      self.message = message
      super().__init__(message)  # pass message to base Exception class
age = int(input("Enter your age: "))
try:
   if age < 18:
      raise AgeError("You are not eligible to vote.")
   else:
      print("You can vote!")


except AgeError as e:
```

```
    print("Error:", e)       # prints the message from __init__()

Output:
Enter your age: 15
Error: You are not eligible to vote.
```

What is happening : Age error is a class and raise creates the object as (e) and passes argument to base exception,

Then the program flows changes, and except AgeError as e block matched and is executed

4.6 Define Multithreading

Multithreading in Python means when **multiple parts of the same program run** at the same time to perform **different tasks simultaneously** within a single program.

◇ **Meaning in simple words:**

It lets your program **do many things at once**, like downloading files, reading data, or printing messages — all happening together instead of one after another.

4.7 List pros and cons of Multithreading

☑ **Pros (Good Things)**

1. **Faster Execution (for I/O tasks)**
   - Multiple threads can work together — e.g., one thread downloads a file while another reads data.
2. **Better Resource Utilization**
   - Threads share the same memory and data of the process, saving system resources.
3. **Improved Performance in Multi-tasking**
   - Tasks like web requests, input/output, or background operations can run simultaneously.

❌ **Cons (Bad Things)**

1. **Not real multitasking** – Because of Python's **GIL (Global Interpreter Lock)**, only one thread runs Python code at a time.
2. **Hard to fix errors** – When many threads run together, it's difficult to find bugs.
3. **Mix-up of data** – If two threads change the same data, results can get wrong.
4. **Too many threads slow it down** – Starting too many threads can make the program slower.

4.8 Create threads using Threading Module

A **thread** is a small part of a program that can run **independently**. In Python, we can create threads using the **threading module**.

◇ **Steps to Create a Thread:**

1. **Import threading module**
2. **Define a function** that the thread will run
3. **Create a Thread object** and pass the function as target
4. **Start the thread** using .start()
5. **Wait for thread to finish** using .join() (optional)

4.9 Create Multiple Threads which perform different tasks

```
import threading
import time
```

```
# Task 1: Print numbers
def print_numbers():


for i in range(5):

         print("Number:", i)
         time.sleep(0.5)  # simulate delay

# Task 2: Print letters
def print_letters():
for ch in ['A', 'B', 'C', 'D', 'E']:
          print("Letter:", ch)
          time.sleep(0.5)  # simulate delay

# Task 3: Print greetings
def greet():
for i in range(3):
       print("Hello from thread!")
       time.sleep(0.5)


# Create threads
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)
t3 = threading.Thread(target=greet)

# Start threads
t1.start()
t2.start()
t3.start()

# Wait for all threads to finish
t1.join()
t2.join()
t3.join()
print("All threads have finished execution!")
```

Ch-5 Design Graphical user Interface and Regular Expressions

5.1 Design a Graphical User Interface using TKinter library

What is a Graphical User Interface:

A **Graphical User Interface (GUI)** acts as a bridge between the **user** and the **Python application**. Instead of interacting directly with the program's code or console, the user interacts with the **GUI elements** such as buttons, menus, and text boxes. The GUI then communicates with the underlying Python application to perform the required tasks.

Tkinter is an in-built Python library used to create **Graphical User Interfaces (GUIs)** and comes along with Python. It contains **Tcl/Tk GUI toolkit** and contains various **classes** that provide an easy way to build windows, buttons, labels, and other interface components in Python.

5.2 Design GUI using different Geometry Managers

```
import tkinter as tk

# Create main window

      window = tk.Tk()

      window.title("Simple GUI")

      window.geometry("250x150")

# Create a label

      label = tk.Label(root, text="Hello, Tkinter!")

      label.pack(pady=20)

# Create a button

      button = tk.Button(root, text="Click Me", command=lambda:    label.config(text="Button
      Clicked!"))

      button.pack()

# Run the GUI

      window.mainloop()
```

General Steps to Create a Clean GUI using Tkinter

Step 1: Import the Tkinter module and give it an alias. import tkinter as tk

Step 2: Create the main window of the GUI by creating an object of the Tk class (main_obj_name).
window = tk.Tk()

Step 3: Set the title of your GUI using the title() method. window("My First GUI")

Step 4: Define the default size of your window using the geometry() method. window.geometry("400x300")

Step 5: Create objects of widgets such as Label, Button, Entry, Canvas, etc., and give appropriate attributes to them.

$$label = tk.Label(window, text = "Welcome\ to\ Tkinter!")$$

Step 6: Arrange the widgets using geometry managers like pack(), grid(), or place().

$$label.pack(pady = 20)$$

Step 7: Finally, run the GUI event loop using the mainloop() method, which keeps the window open and responsive..

$$window.mainloop()$$

5.2 Design GUI using different Geometry Managers

In Tkinter, **Geometry Managers** are used to **arrange and position widgets** in the window. There are **three main geometry managers**:

1. **pack()** –

   pack() arranges the widgets in all **four directions** — top, bottom, left, and right — in a **stack form**.

   - The widget that is defined **first comes first** and stays **near the border of the window**.
   - When packed vertically or horizontally, it **occupies the entire row or column**, so no other widget can come beside it in the same line.
   - **It offers Limited control , no widgets can be overlapped**

Attribute	Description	Example
side	Specifies which side of the window to pack the widget ("top", "bottom", "left", "right")	label.pack(side="left")
padx / pady	Adds horizontal or vertical padding around the widget	label.pack(padx=10, pady=5
ipadx / ipady	Adds internal padding inside the widget	button.pack(ipadx=5, ipady=3)
fill	Expands the widget to fill extra space — can be "x", "y", or "both"	button.pack(fill="x")
expand	If set to True, lets the widget expand to fill any extra space	entry.pack(expand=True)

EXAMPLE:

```
label1 = Label(window, text="Label 1", bg="red", fg="white")

label1.pack(side=TOP, padx=10, pady=10, ipadx=5, ipady=5,fill=X)

window.mainloop()
```

2. **grid()** –

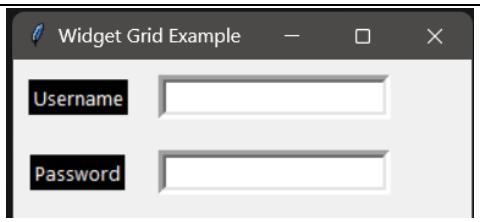
- **grid()** aligns the widgets in a **tabular (matrix-like) format** using **rows and columns**.
- Each widget is placed in a specific **row** and **column**, similar to cells in a table, allowing for a clean and organized layout.
- **It offers medium control , no widgets can be overlapped.**

Attribute	Description	Example
row	Specifies the row number to place the widget	label.grid(row=0)
column	Specifies the column number to place the widget	entry.grid(column=1)
rowspan	Merges the widget across multiple rows	label.grid(row=0, rowspan=2)
columnspan	Merges the widget across multiple columns	button.grid(column=0, columnspan=2)
padx / pady	Adds external (outer) padding around the widget	button.grid(padx=10, pady=5)
ipadx / ipady	Adds internal (inner) padding inside the widget	entry.grid(ipadx=5, ipady=3)

Example:

```
label1.grid(row=0, column=0, padx=10, pady=10)

label2.grid(row=1, column=0, padx=10, pady=10)

entry1.grid(row=0, column=1, padx=10, pady=10)

entry2.grid(row=1, column=1, padx=10, pady=10)

window.mainloop()
```

OUTPUT:



3. place() –it positions widgets at exact x and y coordinates within the window.
   It gives you full control over where each widget appears, unlike pack() or grid() which use automatic layouts.
   Widgets can be overlapped.

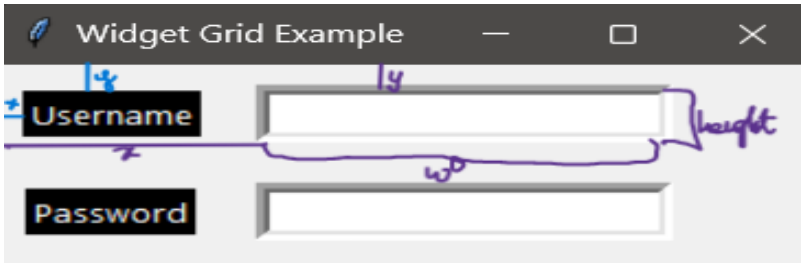
Main Attributes

Attribute	Description	Example

x	Horizontal (X-axis) position in pixels	`label.place(x=50)`
y	Vertical (Y-axis) position in pixels	`label.place(y=100)`
width	Sets the widget's width	`button.place(width=120)`
height	Sets the widget's height	`button.place(height=40)`
relx / rely	Positions widget **relative to window size** (0.0 to 1.0)	`entry.place(relx=0.5, rely=0.3)`
relwidth / relheight	Sets widget's **relative size** to window	`entry.place(relwidth=0.4, relheight=0.1)`

Example:

```
from tkinter import *

window=Tk()

window.title("Widget Grid Example")

window.geometry('300x200')

.
.
.
label1.place(x=20, y=30,width=70,height=20)
label2.place(x=20, y=70,width=70,height=20)

entry1.place(x=100, y=30,width=100,height=30)
entry2.place(x=100, y=70,width=100,height=30)

window.mainloop()
```



❌**You should not mix pack(), grid(), and place() in the same container (like the same window or frame).**

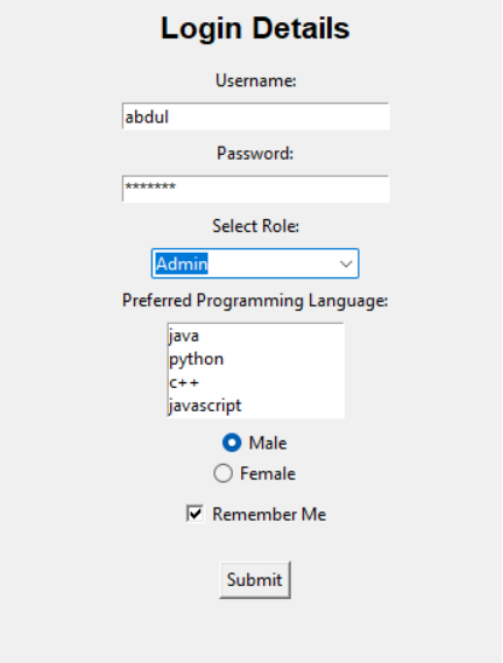
5.3 Use various Widgets in TKinter library

Widget	Purpose / Description	Syntax
Label	A widget that **shows text** or image or **information** on the window.	Label(parent, text="", font=(), bg="", fg="")
Entry	A small **text box** where the user can **type one line** of input.	Entry(parent, width=value, show="")
Text	A bigger text area where users can type **multiple lines**.	Text(parent, height=value, width=value)
Radio button	A round button where the user can choose **only one option** from a group.	Radiobutton(parent, text="", variable=var, value=value, command=function)
Check button	A square button that can be **checked or unchecked and has 2 states 1 or 0**. Used when multi-option can be selected	Checkbutton(parent, text="", variable=var, onvalue=1, offvalue=0, command=function)
Button	Creates a clickable button that triggers a function.	Button(parent, text="", command=function)
Listbox	Displays a scrollable list of items.	Listbox(parent, height=value, selectmode="single")Listbox.insert(END, list_name)
Message box	Shows information, warning, or error popup windows.	messagebox.showinfo("Title","Message") messagebox.showwarning("Title","Warning") messagebox.showerror("Title","Error")
Canvas	Used for drawing shapes, images, lines, and custom graphics.	`Canvas(parent, width=value, height=value, bg="")`
Scrollbar	Adds scrolling ability to Text, Listbox, Canvas, etc.	`Scrollbar(parent, orient=VERTICAL/HORIZONTAL)`

```
import tkinter as tk

from tkinter import messagebox

from tkinter import ttk



# Create main window
```

```
root = tk.Tk()
root.title("Login Details")
root.geometry("400x400")

# ----- Label -----
title_label = tk.Label(root, text="Login Details", font=("Arial", 16, "bold"))
title_label.pack(pady=10)

# ----- Username -----
user_label = tk.Label(root, text="Username:")
user_label.pack()
user_entry = tk.Entry(root, width=30)
user_entry.pack(pady=5)

# ----- Password -----
pass_label = tk.Label(root, text="Password:")
pass_label.pack()
pass_entry = tk.Entry(root, width=30, show="*")
pass_entry.pack(pady=5)

# ----- Combobox (Role Selection) -----
role_label = tk.Label(root, text="Select Role:")
role_label.pack()
role_combo = ttk.Combobox(root, values=["Admin", "User", "Guest"])
role_combo.pack(pady=5)

# ----- Listbox (Preferred Language) -----
lang_label = tk.ttk.Label(root, text="Preferred Programming Language:")
lang_label.pack()
```

```
lang_list = tk.Listbox(root, height=4)

lang_list.insert(tk.END, "java")

lang_list.insert(tk.END, "python")

lang_list.insert(tk.END, "c++")

lang_list.insert(tk.END, "javascript")

lang_list.pack(pady=5)


#Radiobuttons for gender selection

radio_var = tk.StringVar()

male=tk.ttk.Radiobutton(root, text="Male",
variable=radio_var, value="Male")

female=tk.ttk.Radiobutton(root, text="Female",
variable=radio_var, value="female")

male.pack()

female.pack()


# ----- Checkbutton (Remember Me) -----

remember_var = tk.BooleanVar()

remember_check = tk.Checkbutton(root, text="Remember Me", variable=remember_var)

remember_check.pack(pady=5)



# ----- Button Function -----

def submit():

    username = user_entry.get()

    password = pass_entry.get()

    role = role_combo.get()

    selected_lang = lang_list.get(tk.ACTIVE)

    remember = remember_var.get()

    gender=radio_var.get()

# ----- Submit Button -----
```



```
submit_btn = tk.Button(root, text="Submit", command=submit)

submit_btn.pack(pady=15)



# Run GUI

root.mainloop()
```

5.4 List attributes of widgets

Main Widgets and Their Main Attributes

Widget	Main Attributes
Label	text, bg, fg, font, width, height, image, anchor
Entry	width, font, bg, fg, show, state, textvariable
Text	width, font, bg, fg, height, state, wrap
Check-button	Text, bg, fg, font, variable, onvalue, offvalue, command
Radio-button	text ,bg, fg, font, variable, value, command
Button	text, bg, fg, font, command, state, cursor, relief
List-box Listbox.insert()	width,bg, fg, font, height, select-mode, exports-election End, “text”
Message-box	title, message, icon, type
Canvas	bg, width, height, bd, relief, highlight-thickness
Scale	from_, to, orient, length, tick-interval, resolution, variable
Scrollbar	

5.6 Create patterns to use regular expressions

In simple terms

Regex is a **pattern-matching tool** that matches a **given expression** with **predefined pattern rules**, and is used for **searching, extracting, or validating specific text formats** within strings.

- **Creating patterns** = Designing the *rule* (what valid data looks like) to use regex .
- **Validating data** = *Applying* that rule to check if user input is correct.
- **1. Basic Pattern Examples**

Pattern	Meaning	Example Match

abc	Matches the exact sequence of characters “abc”.	It matches “abc” inside the string “123abc456”.
[A-Z]	Matches any one uppercase English letter.	It matches the letter “A” in the word “Apple”.
[a-z]	Matches any one lowercase English letter.	It matches the letter “p” in the word “Apple”.
[0-9]	Matches any single digit from 0 to 9.	It matches the digit “4” in the string “Room4”.
\d	Matches any digit character (same as `[0-9]` in most cases).	It matches the digit “5” in the string “5km”.
\w	Matches any letter, digit, or underscore.	It matches the entire string “name_1” because all characters are word characters.
.	Matches any single character except a newline.	It matches the letter “c” in “cat” or “u” in “cut”.

4 Quantifiers (Repetition)

TOKEN	MEANING
`\D`	Digit (0-9)
`\D`	Not digit
`\W`	Word character (a-z, A-Z, 0-9, _)
`\W`	Not word character
`\S`	Whitespace (space, tab)
`\S`	Not whitespace
*	**0 or more times**
+	**1 or more times**
?	**0 or 1 time**
{N}	**Exactly n times**
{N,}	**At least n times**
{N,M}	**Between n and m times**

TOKEN	MEANING
^	Start of string
$	End of string
\B	Word boundary
\B	Not word boundary

2. Common Real-World Pattern Examples

Goal	Regex Pattern	Description
Email pattern	^[\w.-]+@[\w.-]+\.\w{2,3}$	Matches typical email formats
Phone number pattern	^\d{3}-\d{3}-\d{4}$	Like “123-456-7890”
Date (dd/mm/yyyy) pattern	^\d{2}/\d{2}/\d{4}$	Matches “21/10/2025”
Alphabets pattern	^[A-Za-z]+$	No numbers or symbols allowed
Digits pattern	^\d+$	Only numbers allowed

Explaination for email regex: ^[\w.-]+@[\w.-]+\.\w{2,3}$

Part	Correct Meaning
`^`	Marks **start of the string** (pattern begins here).
`[\w.-]+`	Inside `[]`, it allows **letters, digits, underscore (_), dot (.), or hyphen (-)**. The + means **one or more** of these characters. *(Not "one set," but one or more characters from this group.)*
`@`	The **@ symbol** is required — separates username and domain.
`[\w.-]+`	Matches the **domain name**, made of one or more letters, digits, dots, or hyphens.
`\.`	Matches a **literal dot (.)**, which is compulsory before the domain extension.
`\w{2,3}`	Matches **2 to 3 word characters (letters/digits/underscore)** — usually domain extensions like `com`, `in`, `org`.
`$`	Marks the **end of the string**, ensuring nothing extra comes after.

5.7 Validate data using regular expressions

To validate data using regex expression in python we need

- Import the regex module $import\ re$
- Define the pattern rules in a string example: to check email pattern $= r'^[\w.-]+@[\w.-]+\.\w{2,3}$'$
- Store some example text in a text string
- Use the syntax to search pattern in given data or match entire string exactly with the pattern etc .., Some common re functions are

Function	Purpose
re.match(pattern,text)	• Tries to match the pattern **only at the beginning** of the string. • If the match is found at the start → returns a match object; else → None.
re.search(pattern,text)	Finds the **first occurrence** of the pattern anywhere in the string
re.findall(pattern,text)	Returns **all matches** in a list from the entire string
re.fullmatch(pattern,text)	Ensures the **entire string** matches the pattern exactly

Example: to match the email pattern exactly from the text given

```
import re

# Step 1: Define the pattern
pattern = r'^[\w.-]+@[\w.-]+\.\w{2,3}$'
# Step 2: Input or given text
```

```
text = user.name@gmail.com
if re.fullmatch(pattern, text):
    print("✅ Valid email address")

else:
      print("❌ Invalid email address")
```

CH-6 Data Processing

6.1 Open, close, read, write, append data to files using programs

"A file is a structured collection of data stored in non-volatile memory and managed by a file system."

Files stores various data like text, image etc permanently in an non volatile memory . we use files in python when we want to store some data generated by our python program into permanent non volatile memory

Syntaxto open a file : **var_name** = **open**("example.txt","mode r ")

1. open()= It returns a **file object which is stored in** The file gets opened and inside the block u perform operations on it. When the block ends, Python automatically closes the file — even if an error happens.

2. "example.txt"= Python will look for this file in the **current folder** unless you give a full path.

3. "mode"= It tells Python **in which mode** you want to open the file.

1. Open & Close

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

# work with file

file.close()
```

file.close()

2. Read

```
with open("example.txt", "r") as file:

  data = file.read()

  print(data)
```

- **read() or read(n)** → whole file or n characters
- **readline()** → one line
- **readlines()** → list of all lines

3. Write (overwrite)

```
with open("example.txt", "w") as file:

    file.write("Hello World")
```

- write() → one string
- writelines() → list of strings
- print(..., file=f) → writes with newline

4. Append

```
with open("example.txt", "a") as file:

    file.write("\nNew Line")
```

👍 **Why with is better than file = open(...) ?**

❖ You don't need to call file.close(). It prevents memory leaks, It makes code clean and safe.

flush() is a file method that **forces Python to immediately write buffered data to the file**. (ecet)

TYPES OF FILES <- click it to read

6.2 List modes of opening a file

✅ **Basic Text Modes**

Mode	Meaning	Notes
"r"	Read	File **must exist**
"w"	Write	Creates new file or **overwrites** existing file
"a"	Append	Adds data to **end** of the file
"r+"	Read + Write	File **must exist**
"w+"	Write + Read	Creates new file or **overwrites** existing file
"a+"	Append + Read	Creates file if it **doesn't exist**

✅ **Binary Modes :** Used for **images, videos, PDFs, audio, etc.**

Mode	Meaning	Notes
"rb"	Read binary	File **must exist**
"wb"	Write binary	Creates or **overwrites** file
"ab"	Append binary	Add binary data to end
"rb+"	Read + Write binary	File **must exist**
"wb+"	Write + Read binary	Creates or **overwrites** file
"ab+"	Append + Read binary	Creates if missing

6.3 Delete files and folders

If you want to **delete files, create folders, rename files, check paths**, etc., you must import os.

OS is a **built-in Python module** that lets your program interact with the **Operating System**.

The OS controls:

- Deleting single files
- Deleting empty folder

1. Deleting Files

- First **import the `os` module**.
- Then use **`os.remove()`** to delete a specific file.
- You **do NOT open the file** before deleting.

Example

```
import os
os.remove("example.txt")
```

Meaning

- `os.remove()` **permanently deletes** the file.
- Not safe alone: if the file **does NOT exist**, it gives an **error**.

Safer Way: Check Before Deleting

We check if the file exists using
`os.path.exists("filename")` → returns True if file exists, False if not.

Example

```
import os
if os.path.exists("example.txt"):
    os.remove("example.txt")
else:
    print("file does not exist")
```

2. Delete an EMPTY folder

- To delete only an empty folder you just need to import os module and use its rmdir() method which removes empty directory.

```
import os
os.rmdir("myfolder")
```

3. Delete a Folder That Has Files Inside

- Import the **shutil** module.
- Use **`shutil.rmtree()`** to delete a folder **along with all files and sub-folders** inside it.

Example

```
import shutil
shutil.rmtree("myfolder")
```

Meaning

- `rmtree()` removes the **entire directory**: files, subfolders, hidden items

So it is more powerful (and risky) than `os.rmdir()`.

About the `shutil` Module

- **shutil** stands for **"shell utilities"**.
- It is used for **advanced file and folder operations** that `os` alone cannot do.

Types of files

Files

- **Text-Based Files (no need special software to open)**
  - .txt
  - .html
  - .cfg
  - .ini
  - .conf
- **Binary Files (need special software to open)**
  - .pdf (opens with PDF Reader)
  - .png (opens with Image Viewer)
  - .jpg (opens with Image Viewer)
  - .mp4 (opens with Video Player)
  - .mp3 (opens with Audio Player)
  - .exe (opened by Operating System)
  - .zip (opens with Archive Manager)

<Back to topic