PYTHON

Topics:

1. Basics

    Introduction to Python
    Variables
    Data Types (int, float, string, bool)
    Type Casting
    Input & Output
    Comments

2. Operators

    Arithmetic operators
    Comparison operators
    Logical operators
    Assignment operators
    Bitwise operators
    Membership (in, not in)
    Identity (is, is not)

3. Strings

    String functions (upper, lower, replace, split…)
    Slicing
    String formatting (f-string, format)

4. Decision Making

    if, elif, else
    Nested if
    Short-hand if

5. Loops
    - for loop
    - while loop
    - break, continue, pass
    - Loop else

6. Data Structures
    - List
    - Tuple
    - Set
    - Dictionary
    - List comprehension
    - Dictionary comprehension
    - Set comprehension

7. Functions
    - Creating functions
    - Arguments (default, keyword, variable-length *args, **kwargs)
    - Return values
    - Lambda functions
    - Recursion
    - local & global scope

8. Modules & Packages
    - Import
    - Math module
    - Random module
    - Creating your own module

9. File Handling
    - Read file
    - Write file
    - Append
    - Handling file exceptions

10. Exception Handling
    - try
    - except
    - finally
    - raising errors

11. Object-Oriented Programming (OOP)

    - Class & Object
    - Constructor
    - Inheritance
    - Polymorphism
    - Encapsulation
    - Abstraction

12. Advanced Topics
    - Generators (yield)
    - Decorators
    - Iterators
    - Regular Expressions (regex)
    - Multithreading
    - Virtual environments
    - JSON handling
    - API calls (requests module)

13. Data & Libraries
    - NumPy basics
    - Pandas basics
    - Matplotlib basics

Day-1(dec-4-25)

CREATOR: GUIDO VAN ROSSUM

Python is a **high level programming language**

EX: JAVA, JAVA SCRIPT , GO, C#

High level Programming language means it is easier for humans to **read, write and understandable ,** because it's closer to english and father from machine code.

Why it's called highlevel: It **hides hardware details** like memory, CPU registers, pointers.

Python used to build:
-> Web development (backend)
-> data science and machine learning
-> Automation & Scripting (DevOps, testing, daily tasks)
-> AI/deep learning
-> cybersecurity

Print function: The `print()` function is a built-in Python function that is used to output messages to the console.

```
print("HELLO PYTHON")
print   ("hello")
print(5 * 5)
print(2+4)
print(5/4)    #This is inline comment
print(5%20)  #(5,20 - operands)
print(5-5)    #(*, +, -, % - operator)
print( 10**3)
print (5+2+4)   #(5+2+4 - expression)
print (5+5*5)   #* & + works based on precendence
print(5)    #numbers can print but not words or variables


#build-in functions
print(abs(-10.6))   #absolute value removes the negative sign
print(pow(2,3))
print(max(23,34,25))
print(min(-23,0,3))

#print('HELLO")   #error

#print "HELLO"  #error
```

Variables: A variable is a name that stores a value in Python. Value of a variable can change in run time

RULES:

Must start with letter or underscore

✔ `name`

✔ `_value`

✘ `1name` ❌ **(not allowed)**

Cannot contain spaces

✔ `first_name`

✘ `first name` ❌ `(not allowed)`

Case sensitive

`age` ≠ `Age` ≠ `AGE` ❌ `(not allowed)`

Cannot use Python keywords

✘ `for = 10` ❌ `(not allowed)`

✔ `count = 10`

```
"""
count=4
print(Count)

1num=4
print(1num)
"""

i=1
i=4
print(i)
i=i+1
print(i)

_j=9
print(_j)
```

```
peafowl@peafowl-OptiPlex-3060:~/PYTHON_LEARNING/dec-4$ python3 variable.py
4
5
9
peafowl@peafowl-OptiPlex-3060:~/PYTHON_LEARNING/dec-4$
```

Day-2(Dec-5-25)

Assignment operator:

```
"""
a=10
b=3
a//=b
print(b)
b **=2
print(b)
"""

"""
a = 6
b = 2
c = a + b
c *= b
b **= a
a %= 3
print(a, b, c)
"""


x = 10
y = 3
z = x % y
print(z)
x = x + y
print(x)
~
```

Keywords list : in current release python- 3.12/3.13- Total **35 keywords**

Python Keywords (2025)

Boolean & Null: True, False, None
Conditionals: if, elif, else
Loops: for, while
Loop Control: break, continue, pass
Exception Handling: try, except, finally, raise, assert
Functions: def, return, lambda
Class: class
Modules: import, from, as
Scope: global, nonlocal
Logical: and, or, not
Membership/Identity: in, is
Deletion: del
Async: async, await
Generator: yield

Comments:

EX:
This is a single line comment

EX:
"""
This is a multi-line comment
"""

Built-in DataTypes:

1. **Text:** str
2. **Numeric:** int, float, complex
3. **Sequence:** list, tuple, range
4. **Mapping:** dict
5. **Set:** set, frozenset
6. **Boolean:** bool
7. **Binary:** bytes, bytearray, memoryview
8. **None:** NoneType

1. **Text Type → `str` (String)**
   - Can be single quotes ' ', double quotes " ", or triple quotes.
   - Strings are **immutable** (cannot be changed directly).

     EX: name = "Priya"
     msg = 'Hello!'

2. **Numeric DataTypes:**
   a) **int(Integer)**

   Whole numbers, positive or negative.
   No decimal point.
   EX: x = 10
   age = -5

   b)float

   Numbers with decimal points.(positive, negative, zero)
   EX: price = 99.5
   pi = - 3.14
   a=0.0

   c) complex

   Stores complex numbers with real and imaginary parts.
   EX: z = 2 + 3j

3. **Sequence:**
   a) **List**

   Ordered, Changeable (mutable), Allows duplicates,Use [ ]
   EX: numbers = [1, 2, 3]
   names = ["a", "b", "c"]

   b) `Tuple`

   Ordered, Non Changeable (mutable), Allows duplicates,Use ( )

EX: t = (1, 2, 3)

c) range

Used for generating a sequence of numbers.

EX: r = range(1, 5) # 1, 2, 3, 4

4. Mapping: dict(Dictionary)

Stores data in **key: value** pairs
Ordered, changeable
No duplicate keys
Use { }
EX: student = {"name": "Priya", "age": 25}

5. Set Types:

a) Set

Unordered, No duplicates,Changeable,Use { }
EX: s = {1, 2, 3}

b)frozenset

Same as set
BUT immutable (cannot change)
EX: fs = frozenset([1, 2, 3])

6. Boolean Type: bool

Only two values:first letter should bbe capital
True or False

EX: is_active = True

7. Binary Types:

a) bytes

Immutable bytes sequence.

EX: b = bytes([65, 66, 67])

b) bytearray

Mutable bytes sequence.
EX: ba = bytearray([65, 66])

c) memoryview

Gives memory-level access to bytes without copying.
EX: mv = memoryview(bytes(5))

8. None Type : NoneType

Represents no value or empty.
EX: x = None

Type Casting in Python (Type Conversion)

Type casting means converting one data type to another. Python has **built-in functions** for this:

1) int() → Convert to integer

Removes decimals and converts to whole numbers.

```
int("10")       # 10
int(10.9)       # 10
int(True)       # 1
int(False)      # 0
```

2) float() → Convert to decimal number

```
float("10")     # 10.0
float(5)        # 5.0
float(True)     # 1.0
```

3) **str() → Convert to string**

```
str(10)          # "10"
str(3.14)        # "3.14"
str(True)        # "True"
```

4) **bool() → Convert to True/False**

0, 0.0, "", [], {}, None → False
Everything else → True

```
bool(1)          # True
bool(0)          # False
bool("")         # False
bool("hi")       # True
```

5) **list(), tuple(), set() → Convert between collections**

```
list((1,2,3))        # [1, 2, 3]
tuple([4,5])         # (4, 5)
set([1,1,2])         # {1, 2}
```

Two Types of Type Casting

1)Implicit Type Casting (Automatic)

Python automatically converts one data type to another without your instruction.

- Happens only with **numeric types**.
- Prevents data loss.

EX:

```
x = 5      # int
y = 2.5    # float
z = x + y  # int + float → float
```

Result: `z = 7.5` (Python converts int → float automatically)

2) Explicit Type Casting (Manual)

You convert the data type yourself using functions:

`int()`
`float()`
`str()`
`bool()`
`list(), tuple(), etc.`

EX:

```
a = "10"
b = int(a)   # string → int
```

Python Type Conversion Precedence

Implicit Type Casting Precedence

Lowest-> Highest
(bool → int → float → complex)

1. Bool
2. Int
3. float
4. complex

NOTE: Python only applies **automatic type conversion** (coercion) to numeric types: (int,float,complex)

Explicit Type Casting (manual)

So for explicit casting, there is **no highest or lowest,** because you choose the type.

No precedence

Memory Address:

memory address of a variable using the built-in function `id()`You decide the type manually

```
a = 10
print(id(a))
```

```
x = "hello"
y = [1, 2, 3]

print(id(x))  # memory location of string object
print(id(y))  # memory location of list object
```

Input & Output in Python

1) Input (Taking user input)

Used to read data from the keyboard.

Syntax: input()

EX:

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

2) Output (Printing to the Screen)

Used to display information.

Syntax: print()

EX:

```
print("Hello Python")
print(10)
print("Sum =", 5 + 3)
```

Important Points

a) `input()` always returns data as a string

EX:

```
x = input("Enter a number: ")
print(type(x))   # str
```

b) Convert input to other types using type casting

EX:

```
age = int(input("Enter age: "))
height = float(input("Enter height: "))
```

3) Taking Multiple Inputs

EX:

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

Convert to int:

python

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

4) Output Formatting

EX:

Using commas:

```python
print("Name:", name)
```

Using f-strings (best method):

```python
name = "Priya"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

Using format():

```python
print("Name: {}, Age: {}".format("Priya", 25))
```

For LOOPS:

Syntax: for val in sequence:
#Body of for

Here Val is the variable

range(start, stop)

```
help(range)

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

Print n to n-1

Range is built-in function

range(start,stop,step)

```
for i in range(1,20,3):
        print(i)
~
~
~
~
~
```

```
for i in range(11):
        print(i)

```

```
for i in range(1,11):
    print(f"5*{i}={5*i}")     #f-strings
~
~
```

Indentation : Indentation means giving spaces before a line of code.

```
peafowl@peafowl-OptiPlex-3060:~/PYTHON_LEARNING/dec-8$ python3 forloop.py
  File "forloop.py", line 8
    print(i)
    ^
IndentationError: expected an indented block
peafowl@peafowl-OptiPlex-3060:~/PYTHON_LEARNING/dec-8$
```

How many spaces?

- Standard: 4 spaces
- Tabs not recommended
- All lines in the same block must have same indentation

f-strings (formatted string literals) let you insert variables directly inside a string using { }. They are fast, simple, and readable.Introduced in release 3.6+

Syntax: f"your text {variable}"

Functions: A function is a block of code that runs only when we call it.

Why use functions?

- To avoid repeating the same code
- To make code clean and organized

Function Definition (we write it first)

Syntax: def function_name(parameters): [no spaces, must start with letter or _]

```
# block of code
  return value
```

Function Call (we write it after definition)

EX:

```
def greet():   # ---- Function Definition ----
    print("Hello!")
greet()   # ---- Function Call ----
```

```
def product(a,b):
    result=a-b
    print(result)
    sum=a+b
    print(sum)
    product=a*b
    print(product)

product(3,2)
product(10,0)
```

return: return keyword is used to send output to outside of the function definition

```
def multiply(a,b):
    c=a*b
    print(c)
    return c   # if not use the print(x) is none


x=multiply(3,5)
print(x)
x=2
print(x)
~
~
```

Global variable: A global variable is created outside a function and can be used anywhere.

Created **outside** function

Can be used **inside** and **outside**

Lives until program ends

Local variable: A local variable is a variable inside a function.

Only exist *inside* the function
Cannot be used *outside*
Deleted when function ends

Sum of squares of first n even numbers

```
1 def sum_of_squares(n):
2     total=0
3     for i in range(1,n+1):
4         even=2*i
5         total=total+(even*even)
6     return total
7 print(sum_of_squares(5))
8
```

First five natural means 1 to 5(1,2,3,4,5)
Even = 2*1 , 2*2, 2*3, 2*4, 2*5 (multiply by 2)
total= even *even=total =4+16+36+...=220

Default Arguments: Default arguments are values that a function uses when you do NOT pass a value.

Syntax: parameter = value

EX:

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

 greet()        # No argument → uses default → "Guest"
greet("Priya")  # Argument passed → overrides default
```

Keyword Arguments (Named Arguments): Keyword arguments are arguments where you specify the parameter name when calling the function.

Order does NOT matter
You write parameter = value

EX:

```
def info(name, age):
        print(name, age)
info(age=22, name="Priya")   # order changed → still correct
```

Positional Arguments (Required Arguments): Positional arguments are assigned based on their position (order) in the function call.

Order matters
First value → first parameter
Second value → second parameter
You must give them (required arguments)

EX:

```
def details(a, b, c):   [a=1,b=2,c=3]
        print(a, b, c)
details(1, 2, 3)
```

EX: positional, keyword, default combination

```
def student(name, course="Python", duration="3 months"):
    print(name, course, duration)

student("Lakshmi","JAVA","4 months")  #positional arguments
student(name="priya" ,duration="2 months", course="c")   #keyword arguments
student("Harshini")    #default arguments
```

RULES:

Type	When used	Rules
Positional	When you pass values without naming them	Must come first
Keyword	When you pass key=value	Can be in any order
Default	When value not given	Must be defined after positional parameters in function

Variable-Length Arguments in Python

1. *args [Variable-length Positional Arguments]

- Accepts any number of positional arguments
- Stored as a tuple

```
def add(*args):
    addall = 0
    for num in args:
        addall += num
    print(addall)

add(1, 2, 3)
```

2. **kwargs [Variable-length Keyword Arguments]

- Accepts any number of keyword arguments
- Stored as a dictionary

key : value

"name" : "Priya"

"age" : 25

What kwargs.items() do?

("name", "Priya")

("age", 25)

✔ kwargs = dictionary

✔ kwargs.items() = list of (key, value) pairs

✔ for key, value = unpack each pair

```
def create_profile(**kwargs):
    print("User Profile:")
    for key, value in kwargs.items():
        print(f"{key} = {value}")

create_profile(name="Harshini", city="Hyderabad", role="Tester")
```

Recursion: Recursion means a function calling itself. It continues until it reaches a stopping condition called the base case.

```
def display(n):
    if n == 0:   # base case (stop here)
        return
    print(n)
    display(n - 1)   # function calling itself

display(5)

~
```

Conditional (or) Decision Making Statements:

1. **If statement:** Runs only when the condition is True.

   Syntax:

   if condition:

   Statement

```
age=18
if (age>=18):
    print(True)

~
~
```

2. **If-Else statement:**

   If condition is True → run if block

   Else → run else block

   Syntax:

   if condition:

   statement1

   else:

   statement2

```
age=12
if (age>=18):
    print(True)
else:
    print(False)
~
~
```

3. **If-Elif-Else:** Used when you have multiple conditions.

Syntax:

```
if condition1:
        Statement1
elif condition2:
        Statement2
else:
        Statement3
```

```python
age = 18

if age > 18:
    print("major")
elif age == 18:
    print("equal")
else:
    print("minor")
```

4. **Nested IF (IF inside IF):**

syntax:

```
if condition1:
    if condition2:
        statementA
    else:
        statementB
else:
    statementC
```

```python
age = 25

if age >= 18:
    print("Adult")

    if age >= 60:
        print("Senior Citizen")
    else:
        print("Not senior")
else:
    print("Minor")
```

Logical Operators:

1. **Arithmetic Operators:** Used for mathematical calculations.

   \+ , - , * , / , // , ** , %

```
# +, -, *, / ,**, //, %
a=3
b=5
print(a+b)
print(a-b)
print(a*b)
print(a**b)
print(a//b)
print(a%b)
print(a/b)
~
```

2. **Comparison Operators:** Used to compare values → always returns True or False.

   == , !=, > , <, >=, <=

```
print(5==5)
print(1!=0)
print(5<=25)
print(5>=25)
print(3<2)
print(2>5)
~
```

3. **Logical Operators:** Used to combine conditions.

   and, or, not

```
# and, or, not, xor


print( True and True)   #if both are true only true otherwise false
a=5
print( a<5 and a==5)

print( True or False)   #if any one of them is true , o/p is True othewise false

print( not True)
print( not False)    #if true- o/p false, viceversa


print ( True ^ False)   #if both are different True&false o/p is True, is same False
print ( False ^ True)
print(False ^ False)
print( True ^ True )
```

4. **Assignment Operators:** Used to assign values.

= , += , -= , *= , /=

```
a=5
a+=6
print(a)

a-=2
print(a)

a/=4
print(a)


a*=2
print(a)
```

5. **Bitwise Operators:** Work on bits (0s and 1s).

& , | , ^ , ~ , << , >>

```
# &,|, ^, ~, <<, >>

print(5&3)
print(5|3)
print(5^3)
print(~3)
print(5<<1)
print(5>>1)
print(5<<2)
~
```

6. **Membership Operators:** Used to check whether a value is inside a sequence.

in , not in

```
print(3 in [1, 2, 3])          # True
print("a" in "apple")          # True
print(10 in (5, 8, 10))        # True
print("x" in "hello")          # False


print(5 not in [1, 2, 3])       # True
print("z" not in "hello")       # True
print(10 not in (5, 10, 15))    # False
```

7. **Identity Operators:** Check memory address, not value.

is , is not

a . is : Checks if both variables point to the **same object in memory.**

b. is not: Checks if variables **do NOT refer to the same memory object**.

```
a = [1, 2, 3]
b = a
print(a is b)        # True →both refer to the same list


x = 10
y = 10
print(x is not y)   # False →small integers share memory
```

While Loop: A while loop in Python is used to repeat a block of code as long as a condition is True.

Syntax:

```
while condition:
    # code to repeat
```

EX:

```
i=1
while i<=5:
    print("5")
    i+=1
~
```

NOTE: Python does NOT have a built-in `do…while` loop, but we can simulate it.

Keyword	Meaning	Effect
break	stop loop	exits the loop
continue	skip iteration	goes to next iteration
pass	do nothing	placeholder

break: Stops the loop completely.

Syntax: break

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
"""
```

Continue: skips the current iteration and goes to the next one.

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

Pass: **Does nothing.**

Used as a placeholder when the code is required syntactically but you don't want to write anything yet.

```
for i in range(1, 6):
    if i == 3:
        pass    # does nothing
    print(i)
```

Strings: A **string** is a sequence of characters enclosed in quotes.

Syntax: you can use

- Single quotes ' '
- Double quotes " "
- Triple quotes ''' ''' or """ """ (for multi-line strings)

EX:

```
s1 = "Hello"
s2 = 'Python'
s3 = """This is a multi-line string"""
```

Note: Strings are immutable, You cannot change characters of a string.

EX:

```
s = "python"
s[0] = "P"   # ❌ error [capital]
```

String Functions/Methods: Most imp and commonly used string methods/functions.
These below are **Converting cases form one case to other** :

Method	Use
upper()	uppercase
lower()	lowercase
strip()	remove spaces
replace()	replace text
split()	string → list
join()	list → string
find()	find index
count()	count occurrences
startswith()	prefix check
endswith()	suffix check
isdigit()	only digits
isalpha()	only letters
alnum()	letters + numbers
title()	title case

capitalize() capital the first letter
swapcase() change lower letter <-> upper letters
sorted(str1) [It arranges the items **in increasing (alphabetical) order** and returns] them as a **list**.
EX: “Hello” - ['e', 'h', 'l', 'l', 'o']

String Methods – True/False Checking

`str.isalpha()`	Checks only alphabets	All characters are A–Z or a–z	`"abc".isalpha()` → True
`str.isdigit()`	Checks only digits	All characters are 0–9	`"123".isdigit()` → True
`str.isalnum()`	Alphabets + digits	No special characters	`"abc123".isalnum()` → True
`str.islower()`	Lowercase check	All characters are lowercase	`"hello".islower()` → True
`str.isupper()`	Uppercase check	All characters are uppercase	`"HELLO".isupper()` → True
`str.isspace()`	Space check	Only whitespace (space, tab, newline)	`" ".isspace()` → True
`str.istitle()`	Title case check	First letter of each word is capital	`"Hello World".istitle()` → True
`str.startswith()`	Starts with substring	String begins with given value	`"python".startswith("py")` → True
`str.endswith()`	Ends with substring	String ends with given value	`"hello.txt".endswith(".txt")` → True
`str.__contains__()` or `"in"`	Contains substring	Substring is present	`"apple" in "pineapple"` → True

Modules in python:

A Python file (.py) that contains code — like variables, functions, classes — which you can reuse in other programs.

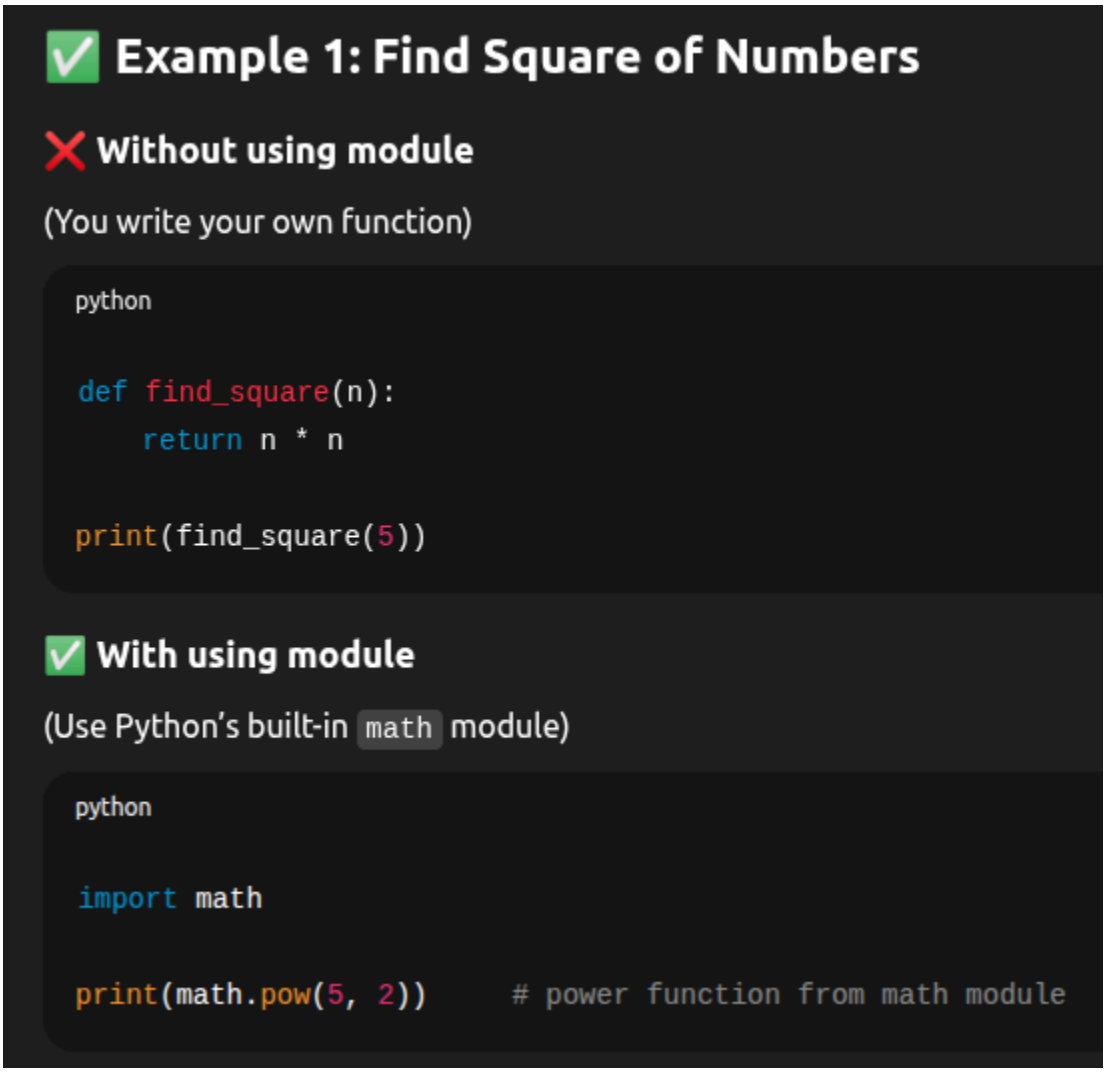
EX:

math.py → module
string.py → module
random.py → module

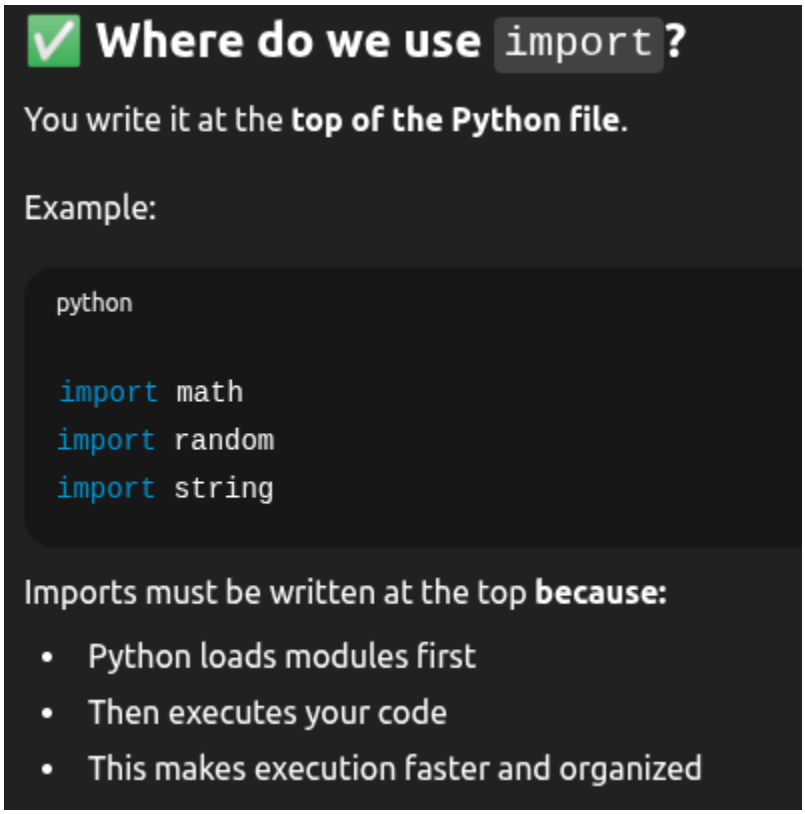
Why used:

Modules:

- Reduce your code
- Are well tested
- Make programs readable
- Provide powerful tools (math, random, os, sys, datetime…)



Import: import is used to bring extra features into your Python program. (or) bring external code (module) into your program.



❌ Without import

You write everything yourself:

```python
# Find area of circle (you write the for
pi = 3.14159
r = 5
area = pi * r * r
print(area)
```

✅ With import

Use ready-made values/functions:

```python
import math

r = 5
area = math.pi * r * r
print(area)
```

ASCII VALUE in STRING: Use the ord( ) function.

```
str1="A"
str2="B"
print(str1>str2)

print(ord(str1))
print(ord("z"))
~
```

OOPS: Object-oriented programming language:

Object-Oriented programming means organizing code around “objects” that have data and behavior, just like real-world things.

Data is attributes, Behaviour is methods

Object-Oriented Programming (OOP) = Think of your program as real-world objects.
Each object has its own data (attributes) and actions (methods).
This makes programs organized, reusable, and easy to understand.

Attributes → Data about an object (like properties or characteristics)
To store information about the object
Each object can have different values

Methods → Actions the object can do (like functions)
To perform actions related to the object
Instead of writing separate functions, methods belong to the object

Attributes = Who/What it is
Methods = What it can do

EX:

```
class Car:
    def __init__(self, color, speed):  # Attributes
        self.color = color
        self.speed = speed

    def drive(self):                # Method
        print(f"{self.color} car is driving at {self.speed} km/hr")

# Object with data and action
my_car = Car("Red", 100)
my_car.drive()
```

Feature	Structured Programming	OOP
Focus	Functions	Objects
Data	Separate	Inside objects
Reusability	Less	High
Security	Low	High (Encapsulation)
Best for	Small programs	Large applications

Class: class is used to define a class, which is the blueprint for creating objects.

class is a keyword

You cannot use class as a variable name

Syntax:

class Country: [country is a class name]

pass

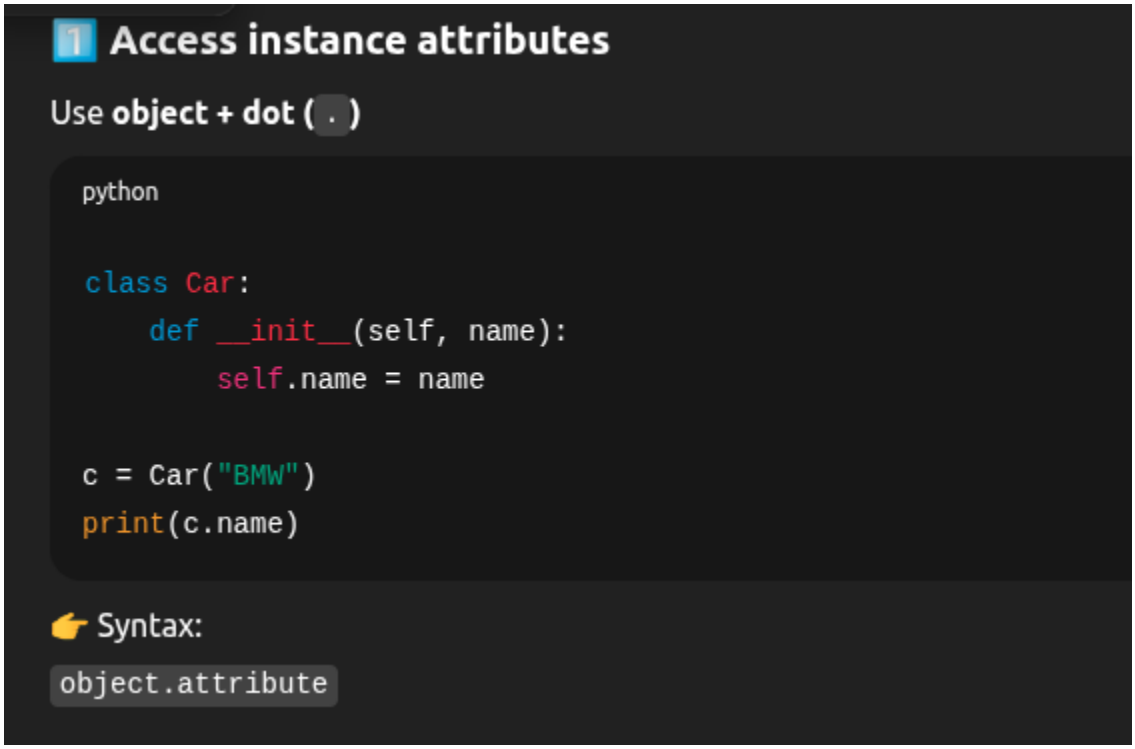
```python
class Car:
    wheels = 4              # class attribute

    def __init__(self, name):
        self.name = name  # instance attribute
```

Types:

Instance attributes → belong to object

c**lass attributes** → shared by all objects

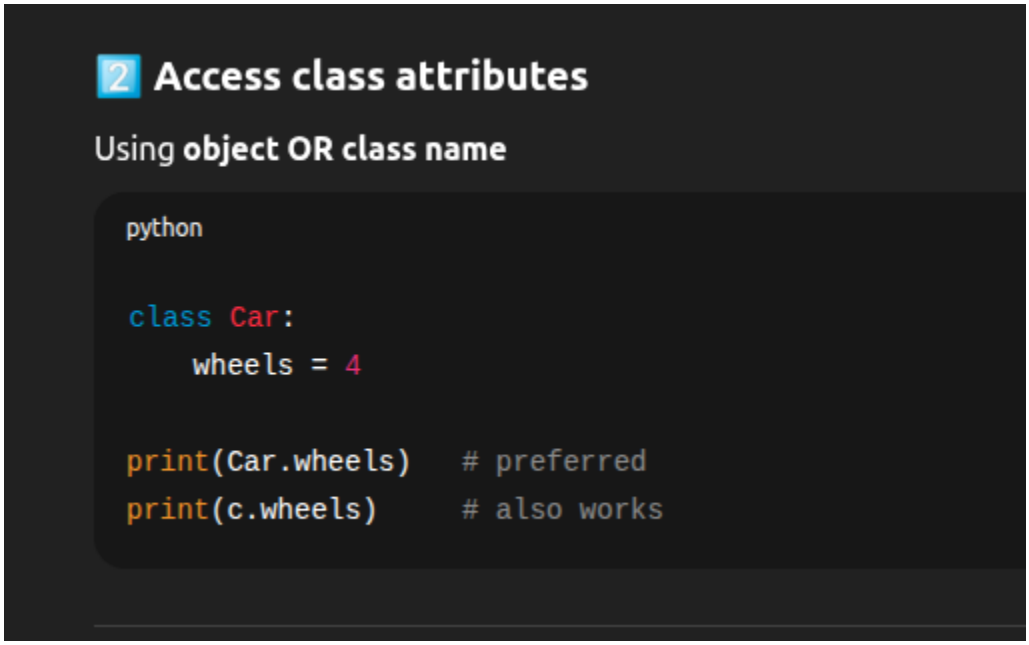
Access that instance attributes



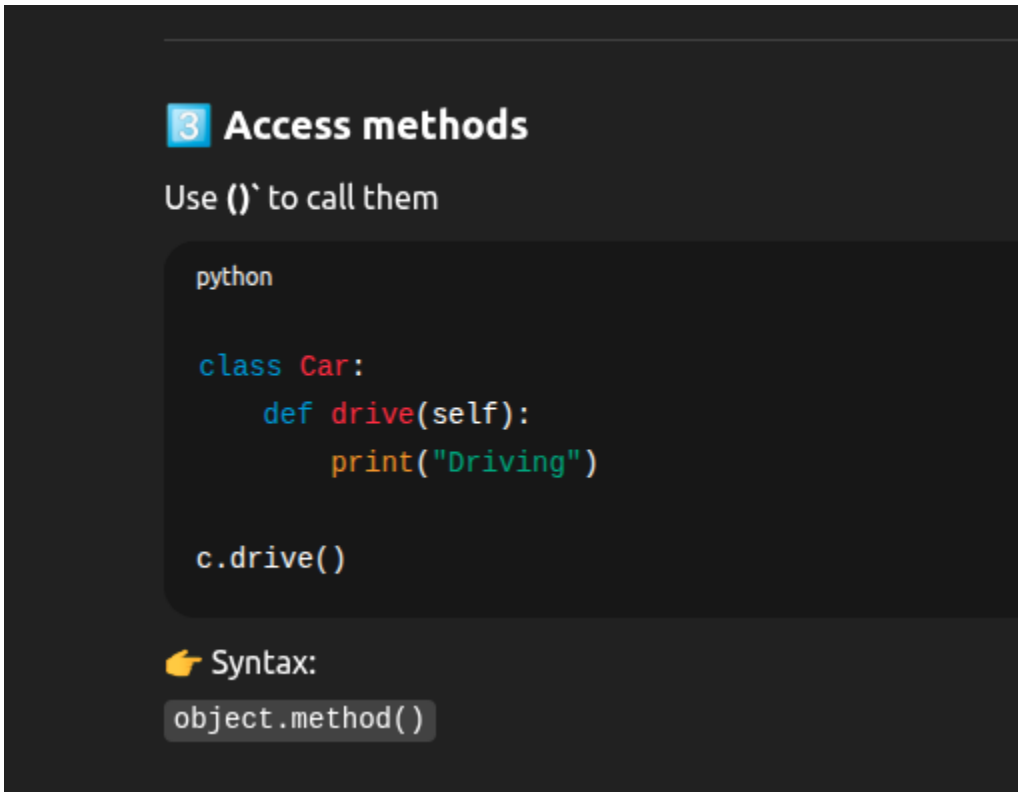
EX:

print(car1.color) # Red / print(car2.speed) # 150

Access class attributes



Access methods



Objects or instance: An instance is a specific object created from a class, stored in memory**.** actual thing created using that blueprint.

An object is created by calling the class name like a function.

Syntax:

object_name = ClassName(arguments)

EX: car1 = Car("Red", 120)

Once object is created we can access methods and variables of a class

Calling object methods

car1.drive( )

Python internally does:

Car.drive(car1)

car1 is passed automatically as self

That's why we don't write self while calling

```
class toys:
    doll="barbie"
    ball_colour="white"

access=toys()   #object is access , toys is classname

print(access.doll)
print(access.ball_colour)
~
~
~
~
```

Constructor: In Python, a constructor is implemented using the __init__ method.

Why use __init__?

- __init__ is a special method
- Python automatically calls it when an object is created
- It is used to initialize (set) object data

`self` is used to refer to the current object and access its data and methods.

When a method is always inside the class we use `self`

NOTE: Python does NOT support multiple constructors directly

- Python only keeps the last __init__
- So the first one is lost

```
class Human():
    def __init__(self):
        print("constructor")   #once object is created first constructor will call and execute

priya=Human()
~
~
~
~
-- INSERT --
```

O/P:

```
peafowl@peafowl-OptiPlex-3060:~/PYTHON_LEARNING/dec-16$ python3 class.py
constructor
peafowl@peafowl-OptiPlex-3060:~/PYTHON_LEARNING/dec-16$
```

Syntax:

```
class ClassName:
    def __init__(self, parameters):
        self.variable = value    #create attribute
        print(self.variable)     #uses attribute
```

Call Flow:

Class created → object created → __init__ called → data stored → accessed

```
#1 Class creation
class Student:
  #2 Constructor (special method) definition inside class
  def __init__(self, name, age):
    #3 Object attribute creation inside constructor
    self.name = name   # object attribute
    self.age = age     # object attribute

#4 Object creation (constructor called automatically)
s1 = Student("Ravi", 20)

#5 Accessing object attributes
print(s1.name)  # prints "Ravi"  # accessing object attribute
print(s1.age)   # prints 20       # accessing object attribute
```

NOTE: **Use `.` operator whenever you are accessing or updating an object's data. [can't access direct]**

Accessing data:

```
print(p1.age)     # 25
print(p1.name)    # Priya
```

Updating data

```
p1.age = p1.age + 1
print(p1.age)     # 26
```

WRONG: p1 = p1 + 1 # ❌

ENCAPSULATION: Binding data and methods into a single unit means combining variables and functions together inside a class.

```
class Student:
    def __init__(self, name, marks):
        self.name = name       # data
        self.marks = marks     # data

    def display(self):        # method
        print(self.name, self.marks)
```

Encapsulation means **hiding internal details of a class and only exposing what's necessary.** It helps to protect important data from being changed directly and keeps the code secure and organized.

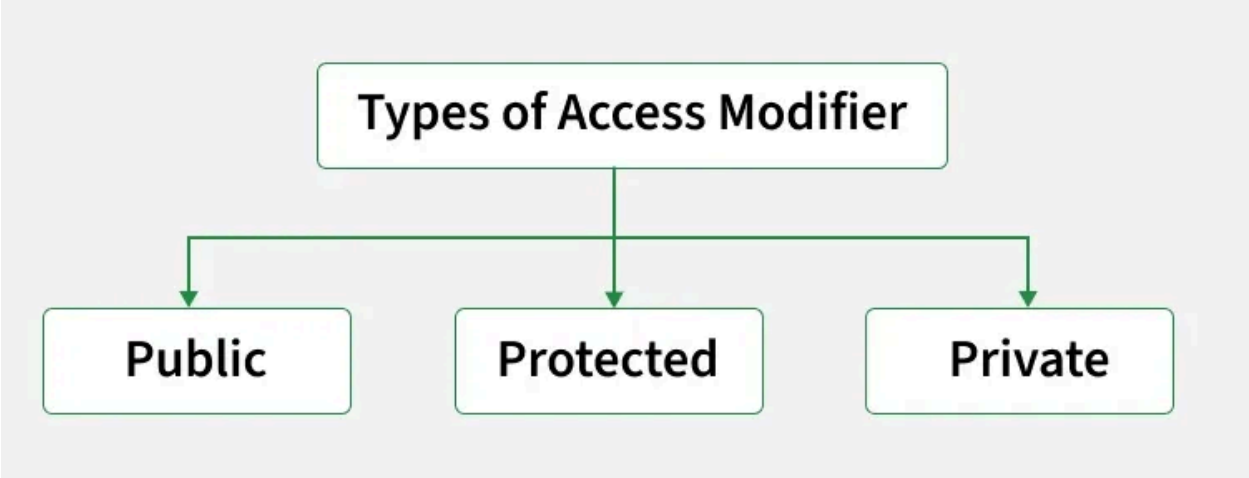
Encapsulation uses mainly:

1. Data hiding
2. Security
3. Controlled access
4. Code maintainability

__repr__ -> string representation of an object

`__repr__` is a **special (magic / dunder) method**, not a keyword.

Access Specifiers (Access Modifiers): Access specifiers define how class members (variables and methods) can be accessed from outside the class. They are 3 types:



1. **Public:** Public members are variables or methods that can be accessed from anywhere.

   Meaning

   Accessible **everywhere**
   Inside class
   Outside class
   Other classes

By default, all members in Python are public.They are defined without any underscore prefix.

Syntax: self.name , emp.name [no underscore]

EX:

```
class Student:
    def __init__(self, name):
        self.name = name   # public

s = Student("Priya")
print(s.name)   # ✅ allowed
```

2. **Protected:** Protected members are variables or methods that are intended to be like below

Meaning

1. Accessible **inside class**
2. Accessible in **child class [subclass]**
3. Should not be accessed outside (by convention)

Syntax: self._name , self._age [single underscore]

Protected members should not be accessed outside the class hierarchy, but Python does not enforce this rule strictly.

EX:

```
class Employee:
    def __init__(self, name, age):
        self.name = name       # public
        self._age = age        # protected

class SubEmployee(Employee):
    def show_age(self):
        print("Age:", self._age)   # Accessible in subclass

emp = SubEmployee("Ross", 30)
print(emp.name)        # Public accessible
emp.show_age()         # Protected accessed through subclass
```

3. **Private:**Private members are variables or methods that cannot be accessed directly from outside the class. They are used to restrict access and protect internal data.

Meaning

1. Accessible **only inside the class**
2. Not accessible outside or in child class

Syntax: self.__salary [double underscore]

Ex:

```python
class Employee:
    def __init__(self):
        self.__salary = 40000

    def get_salary(self):
        return self.__salary

e = Employee()
print(e.get_salary())   # ✅
# print(e.__salary)     # ❌ error
```

EX: public &private method

```python
class Demo:
    __a = 5         # private variable
    b = 100         # public variable

    def __display(self):   # private method
        print(self.__a)

    def show(self):        # public method
        self.__display()  # calling private method inside class

obj = Demo()
print(obj.b)    #  public variable access
obj.show()      # public method → private method
```

In private method [__display] can't access outside the class , if want to access we have to write one public method inside call private method

INHERITANCE: Inheritance is an OOP concept where a child class inherits properties and methods from a parent class.

Parent class = base class [have both attributes and methods]
Child class = derived class [have both attribute and methods]

By creating object in child class , we can get both child class attributes & methods as well as parent class attributes and methods [but not works in viceversa]

“Inherits” means: **Gets / receives / takes properties and methods from another class.**

Syntax:

class ChildClass(ParentClass): [create object only for childclass]

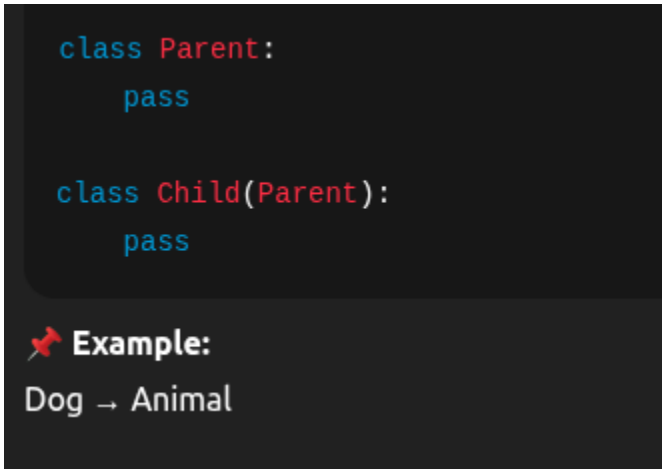
EX:

class Dog(Animal):

TYPES OF INHERITANCE:

1. Single Inheritance: One child inherits from one parent

Syntax:



EX:

```
class Baseclass:
    a=10
    b=100
    def display(self):
        print("Base class")

class DerivedClass(Baseclass):    #inheritance applied when it created like this, no need to create 2nd object
    c=20
    d=30
    def show(self):
        print("Derived Class")

baseobject=DerivedClass()  #base class object
print(baseobject.a, baseobject.b, baseobject.c, baseobject.d)
baseobject.display()
baseobject.show()
```

2. Multiple Inheritance: One child inherits from multiple parents

Syntax:

```
class Father:
    pass

class Mother:
    pass

class Child(Father, Mother):
    pass
```

📌 **Example:**

Child → Father + Mother

EX:

```
class Father:
    def fdisplay(self):
        print("FATHER CLASS")
class Mother:
    def mdisplay(self):
        print("MOTHER CLASS")
class Child(Father,Mother):
    def cdisplay(self):
        print("CHILD CLASS")

c=Child()
c.cdisplay()
c.mdisplay()
c.fdisplay()
```

```
CHILD CLASS
MOTHER CLASS
FATHER CLASS
```

3. Multilevel Inheritance: Child inherits from a class which already inherited another class

Syntax:

```
class Grandparent:
    pass

class Parent(Grandparent):
    pass

class Child(Parent):
    pass
```

📌 **Example:**

Grandfather → Father → Son

EX:

```
class GrandParent:
    def gpdisplay(self):
        print("GRAND PARENT METHOD")
class Parent(GrandParent):
    def pdisplay(self):
        print("PARENT METHOD")
class Child(Parent):
    def cdisplay(self):
        print("CHILD DISPLAY")

c=Child()
c.cdisplay()
c.pdisplay()
c.gpdisplay()
```

4. Hierarchical Inheritance: Multiple children inherit from the same parent

Syntax:

```
class Parent:
    pass

class Child1(Parent):
    pass

class Child2(Parent):
    pass
```

📌 **Example:**

Animal → Dog, Cat

5. Hybrid Inheritance: Combination of two or more inheritance types

syntax:

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass
```

📌 **Example:**

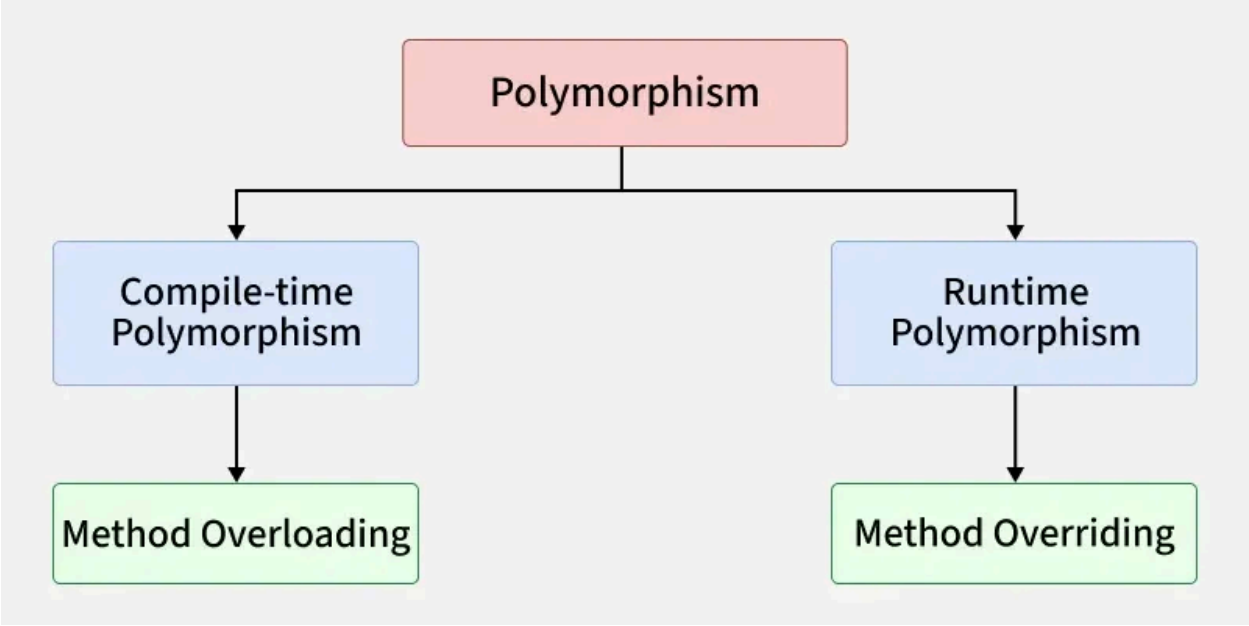
Combination of multiple + hierarchical

super().__init__() is a Python command used within a child class's __init__ method (constructor) to call the __init__ method of its parent (superclass)

POLYMORPHISM: Polymorphism means - “One name, many forms”

Polymorphism is used to write flexible, reusable code where the same method works differently for different objects.

Types of Polymorphism:



1. **Method Overloading (name same, parameters different):**

   NOTE: In Python does not support true method overloading, but we simulate it. [though similar behavior can be achieved using **default or variable arguments.**]

   EX: 1

```python
class Math:
    def add(self, a, b, c=0):
        return a + b + c
```

```python
m = Math()
print(m.add(2, 3))
print(m.add(2, 3, 4))
```

✔ Same method name

✔ Different behavior

EX: 2

```
]: class Demo:
       def add(self,a,b,c=100):
           print(a+b+c)
   obj=Demo()
   obj.add(100,200)
   obj.add(100,200,300)
```

2. **Method Overriding: Child class changes parent class method**

   same method, same parameters

```
class Parent:
    def tranport(self):
        print("cycle")

class child(Parent):
    def tranport(self):
        print("bike")

c=child()
c.tranport()   #override parent method , child will run
```

ABSTRACTION: Abstraction is the process of hiding implementation details and showing only essential features. [uses inheritance]

Python provides abstraction using:

1. Abstract classes
2. Abstract methods

   (from abc module)

"Abc- abstract base class"

Abstract Base Class (ABC) in Python is used to achieve **abstraction** by defining a **common interface** for its subclasses.
An ABC **cannot be instantiated directly** and acts as a **blueprint** for derived classes.

Abstract classes are created using the `abc` module and the `@abstractmethod` decorator.
They **force child classes to implement required methods**, ensuring consistency, while **hiding implementation details**

syntax:

1. Import abc module

   ```
   from abc import ABC,abstractmethod
   ```

2. Create Abstract class

   ```
   class ClassName(ABC):
   ```

3. Define Abstarct methods

   ```
   @abstractmethod
   def method_name(self):
     Pass
   ```

4. Create a child concrete class

   ```
   class ChildClass(ClassName):
   ```

```python
from abc import ABC,abstractmethod
class Abstractdemo(ABC):                    #abstract class
    @abstractmethod
    def Housinginterest(self):          #abstract method
        None
    @abstractmethod
    def Vehicleinterest(self):
        None

class SBI(Abstractdemo):                    #concrete classs, means no abstarct even one also
    def Housinginterest(self):              # should be same method in conncrete and abstarct othe
        print("8.5 interest")
    def Vehicleinterest(self):
        print("5.5 interest")

sbiobj=SBI()
sbiobj.Housinginterest()
sbiobj.Vehicleinterest()
```

Data Structures: A data structure is a way to store, organize, and manage data efficiently so we can access and modify it easily.

EX: without data structure

marks1=40 ,marks2=50, marks3=60

With data structure

marks=[40,50,60……..]

Built-in Data Structures:built-in data structures, each with distinct properties regarding order and mutability:

- List
- Tuple
- Set
- Dictionary

Advanced and User-Defined Data Structures

- Arrays
- Stacks
- Queue
- Linked Lists, Trees, Graphs
- Heaps

Lists: Lists are just like the arrays , List can store both same and different data types.

List = Ordered + Changeable + Duplicates allowed

Ordered [Elements are stored in a **fixed order**, and that order is preserved]

```
nums = [10, 20, 30]
print(nums[0])   # 10
print(nums[1])   # 20
```

Mutable [You can **modify the list after it is created**]

```
nums = [1, 2, 3]
nums[1] = 20
print(nums)   # [1, 20, 3]
```

✔ Add elements
✔ Remove elements
✔ Update elements

Allows duplicates [The **same value can appear multiple times**]

```
nums = [1, 2, 2, 3]
print(nums)
```

Length is calculated using the **number of *elements*** in the list**,** NOT the index.

```
nums = [10, 20, 30, 40]
```

- **Elements** = 4 → `len(nums) = 4`
- **Indexes** = 0, 1, 2, 3 (start from 0)

append() **adds the element to the END (last) of the list.**

list_name.append(element)

insert() **Adds an element at a specific position (index) in the list.**

list_name.insert(index, element)

remove() Removes first occurrence of the value [error if not found value]

```
list_name.remove(value)
```

reverse() Method (changes original list)

```
list_name.reverse()
```

clear() Removes all elements from the list

```
list_name.clear()
```

copy() Creates a shallow copy of the list

```
new_list = list_name.copy()
```

count() Counts how many times a value appears in the list

```
list_name.count(value)
```

sort() sorts the list in ascending order by default

```
list_name.sort()
```

pop() Removes element by index and Returns the removed element

```
list_name.pop(index)
```

extend() Adds multiple elements to the end of a list.

```
list_name.extend(iterable)

list_name+=[ ]      #other way to add end of list
```

del delete by index and by slicing

```
del list_name                    #entire list deleted

del list_name[indexnumber]

del list_name[start: end]
```

Elements from the start index up to (but NOT including) the end index will be deleted.

`start` index → included
`end` index → excluded

Negative indexing removes the last element in the list [-1] last one, [-2], second last ……

2D List:

2dlist= [list1, list2, list3]

EX: [[1,2,3,4] , [5,6,7,8] , [9,10,11,12]]

`ord()` - The function returns the Unicode (ASCII) value of a single character.

syntax: ord(character)

EX: ord('A') # 65

ord('a') # 97

ord('0') # 48

ord('@') # 64

`chr()`- The `chr()` function converts a Unicode (ASCII) value into its character.

syntax: chr(number)

EX: chr(65) # 'A'

chr(97) # 'a'

LIST SLICING:

syntax: list[start : end : step]

EX: number[3:7] start at 3 end at index7

A[3:] start at 3 upto last index
B[:5] start at 0 and end with index5

c[1:5:2]

D[: :-1] reverse order

Tuples: Tuples are similar to lists, but **once created, they cannot be changed**.

Tuples= ordered + immutable (not changeable)+ Duplicates allowed

EX: t = (1, "apple", 3.5, "apple")

Sets: A set is a collection of unique elements. Can store **same or different data types**

Set → unordered, changeable, no duplicates

EX: s = {1, "apple", 3.5, "apple"}

print(s)

```
numbers=[1,2,3,4,2,1]
print(numbers)

numbers_set=set(numbers)
print(numbers_set)

numbers_set.add(4)
numbers_set.add(0)
print(numbers_set)

print(max(numbers_set))
print(len(numbers_set))
print(sum(numbers_set))
numbers_set.remove(4)
print(numbers_set)
print( 1 in numbers_set)


"""
In a set we can perform union, intersection, difference operations
union check by ->  | operator   (combine both set elements)
Intersection check by ->  & operator (elements both in a and b sets)
difference check by ->  - operator  (elemnts prsent in a not in b is difference)
"""
numbers_1_to_6_set1=set(range(1,6))
print(numbers_1_to_6_set1)

numbers_4_to_11_set2=set(range(4,11))
print(numbers_4_to_11_set2)


print(numbers_1_to_6_set1 | numbers_4_to_11_set2)
print(numbers_1_to_6_set1 & numbers_4_to_11_set2)
print(numbers_1_to_6_set1 - numbers_4_to_11_set2)   # A-B
print(numbers_4_to_11_set2 - numbers_1_to_6_set1)   # B-A
~
~
```

`set.intersection(*sets)` → **intersection of multiple sets**

Set comprehension: This is a set comprehension.

Syntax: {expression for item in iterable if condition}

Explanation :

- **expression → what you want to store in the set**
- **item → variable**
- **iterable → list, tuple, range, etc.**
- **condition → optional filter**

EX: {x**2 for x in range(1, 6) if x % 2 == 0}

Dictionaries: A dictionary stores data in key : value pairs.

Dictionary → key–value pairs, ordered, changeable, unique keys[no repeated keys]

Keys → **must be unique (only keys)**
Values → **can be duplicated (only values)**
`items()` → **key + value**

Syntax:

```
dictionary_name = {
    key1: value1,
    key2: value2,
    key3: value3
    }
```

##########Iterating a dictionary

```
for key,value in numbers.items():
    print(f"{key} {value}")
```

`items()` returns **all key–value pairs** from a dictionary.

syntax: dict.items()

EX: student.items()

dict_items([('name', 'Alice'), ('age', 20), ('grade', 89.5)])

EX:

```
numbers= dict(a=5,b=6,c=8, d=9)
print(type(numbers))
print(numbers)

########### Access and Modify Values
numbers['d']=10
print(numbers)
print(numbers['a'])


########## Handling non existing keys
#print(numbers['e'])

#To avoid use get() method, provide a default value if needed
print(numbers.get('e',10))


########## Explorint dictionary methods
print(numbers.keys())
print(numbers.values())
print(numbers.items())


########## Iterating a dictionary
for key,value in numbers.items():
    print(f"{key} {value}")


########## DEL a dictionary
numbers['a']=0
del numbers['a']
print(numbers)
```

Dictionary Comprehension:

syntax: {key: value for item in iterable if condition}

EX: squares = {x: x**2 for x in range(1, 6)}

List Comprehension:

Syntax: [expression for item in iterable if condition]

EX: squares = [x2 for x in numbers]**

Tuple comprehension does NOT exist

( ) creates a generator, not a tuple.

Empty list, tuple, set , dictionary can be denoted as:

List - [ ]

Tuple - ( )

Dictionary - { }

Set - set( )

`enumerate()` is used when you want **both the index and the value** while looping through a list (or any iterable)

syntax: enumerate(iterable, start=0).

EX: fruits = ["apple", "banana", "cherry"]

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

`isinstance()` is used to **check the type of a value**.

syntax: isinstance(object, data_type)

EX:
```
print(isinstance(5, int))       # True
print(isinstance("hi", str))    # True
print(isinstance(3.5, int))     # False
```

Time Complexity: Time Complexity tells how fast an algorithm runs as input size (n) increases.

It is written using **Big-O notation - order of n -O(n)**

Common Time Complexities

Big-O	Meaning	Example
O(1)	Constant time	Accessing list element `a[0]`
O(n)	Linear time	Loop through a list
O(n^2)	Quadratic time	Nested loops
O(log n)	Logarithmic	Binary search
O(2^n)	Exponential	Recursive Fibonacci

Example

As elements increase, the running time changes in a specific order — that order is time complexity.

Example

1️⃣ O(1) – Constant time

Input increases ❌ → Time stays same

python Copy code

```python
print(arr[0])
```

2️⃣ O(n) – Linear time

Elements increase ⬆ → Time increases proportionally

python Copy code

```python
for i in arr:
    print(i)
```

3️⃣ O(n²) – Quadratic time

Elements increase ⬆⬆ → Time increases much faster

python Copy code

```python
for i in arr:
    for j in arr:
        print(i, j)
```

TYPES OF ERROR

There are mainly two types of errors in python programming. Let us learn about both types of python errors:

1.Syntax Errors or Compile Time Error

2.Runtime Errors or Exceptions

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(system error) or a Python exception(runtime error).

Exception Handling: Exception handling prevents program crash by managing runtime errors.

Exception = runtime error that interrupts program execution.

EX:

- Without exception handling → crash [car stops suddenly]
- With exception handling → safe execution [brakes applied safely]

These 5 blocks together help handle errors safely and cleanly in Python.

What happens

Block	When it runs
`try`	Code that may fail
`except`	When error occurs
`else`	When no error
`finally`	Always
`raise`	Manually create error

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num

except ZeroDivisionError:
    print("Cannot divide by zero")

except ValueError:
    print("Invalid input")

else:
    print("Result:", result)   # runs only if no error

finally:
    print("Execution completed")  # always runs
```

EX:

```
peafowl@peafowl-OptiPlex-3060:~/PYTHON_LEARNING/dec-26$ python3 execption-handling.py
Traceback (most recent call last):
  File "execption-handling.py", line 2, in <module>
    j=i/0
ZeroDivisionError: division by zero
peafowl@peafowl-OptiPlex-3060:~/PYTHON_LEARNING/dec-26$
```

File Handling: File handling allows Python to read data from files and write data into files.

Create, open, write, read, delete, append, close…etc

Mode	Meaning	Use
'r'	Read mode	File read cheyyadam
'w'	Write mode	File create / overwrite
'a'	Append mode	Existing file lo add cheyyadam
'r+'	Read and write	
'w+'	Write and read (overwrite)	

1) **Open a file: open(file_path, access_mode) , file1 is variable to store**

```
file = open("file.txt", "mode")
```

2) **Create a file:**

```
open("file.txt", "x")
```

3) **Write to a file:**

```
file = open("file.txt", "w")

file.write("Hello")

file.close()
```

4) **Read a file:**

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

print(file.read())

file.close()
```

5) **Append a file:**

```
file = open("file.txt", "a")

file.write("\nNew line")

file.close()
```

6) **Close a file:**

```
file.close()
```

7) **Rename a file:**

```
import os

os.rename("old.txt", "new.txt")
```

8) **Delete a file:**

```
import os

os.remove("file.txt")
```

EX:

```
s= open("hello.txt", "r")
print(s.read(5))



"""s1= open("/home/peafowl/priya.txt", "r")
print(s1.read(9))
"""

"""s= open("hello.txt", "r")
print(s.readline())
print(s.readline())
"""


"""s= open("hello.txt", "w")
s.write("lakshmipriya")
s.close()
s= open("hello.txt", "r")
print(s.read())
"""


"""s= open("hello.txt", "a")
s.write("lakshmipriya")
s.close()
s= open("hello.txt", "r")
print(s.read())"""
```

Algorithms : An algorithm is a step-by-step method to solve a problem.

1. Linear Search - Linear search checks **each element one by one** until the value is found.

   It works in unsorted list

   1️⃣ Start from first element

   2️⃣ Compare with target

   3️⃣ If match → Found

   4️⃣ If end → Not Found

2. Binary Search- Binary search finds an element by **repeatedly dividing the list into halves**.
   NOTE: 👉 Binary search works only on SORTED data

   1️⃣ Take middle element

   2️⃣ If middle == key → Found

   3️⃣ If key < middle → search left half

   4️⃣ If key > middle → search right half

   5️⃣ Repeat until found or range ends

Data Structures - Stacks and Queues:

1.stack : A stack is a data structure that follows

👉 LIFO – Last In, First Out

Stack Operations

1️⃣ Push (add item)
2️⃣ Pop (remove item)
3️⃣ Peek (top item)
4️⃣ Is Empty

2. Queue: A queue is a data structure that follows

👉 FIFO – First In, First Out

Queue Operations

1️⃣ Enqueue (add element)
2️⃣ Dequeue (remove element)
3️⃣ Front (peek)
4️⃣ Is Empty