PYTHON

Introduction to Python

Python is a high-level, interpreted, and general-purpose programming language. Created by Guido van Rossum and released in 1991, it emphasizes code readability using indentation. Python is dynamically typed and supports multiple programming paradigms.

Example:

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

Python Features

- Easy to learn and use
- Interpreted language
- Dynamically typed
- Rich standard library
- Cross-platform compatibility
- Object-oriented and functional
- Extensible and embeddable
- Strong community support

Python Syntax Basics

- **Comments**

```
# Single line comment
"""
Multi-line comment
"""
```

- **Indentation**: Essential for defining blocks (i.e function, control structure etc)
- **Variables**: Dynamically typed

```
name = "Alice"
age = 25
```

- **Syntax:**

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

  - value1, value2,... → Things you want to print
  - sep (optional) → Separator between multiple values (default is space ' ')
  - end (optional) → What to print at the end (default is new line \n)

Example:

```
print("Python", "is", "awesome", sep="-", end="!!!\n")
```

Output:	Python-is-awesome!!!

Operators

- Arithmetic: +, -, *, /, //, %, **
- Assignment: =, +=, -=, etc.
- Comparison: ==, !=, >, <, >=, <=
- Logical: and, or, not
- Membership: in, not in
- Identity: is, is not

REPL in Python :- Read–Eval–Print Loop

1. **Read:** Takes the input from the user.
2. **Eval:** Evaluates or executes the input as a Python expression.
3. **Print**: Displays the result of the evaluation.
4. **Loop:** Goes back to step 1 and waits for the next input.

Example of using REPL:

You can open REPL by just typing python in your terminal or command prompt (if Python is installed).

You'll see something like:

```
>>> 2 + 3
5
>>> print("Hello")
Hello
>>> a = 10
>>> a * 2
20
```

Each time you enter something, Python:

- Takes the input
- Executes it
- Shows the result
- Waits for the next command

Control Structures

If-Else:

```
if age >= 18:
    print("Adult")
elif age >= 13:
    print("Teenager")
else:
    print("Child")
```

Loops:

- **For Loop:**

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

- **While Loop:**

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

- **Break and Continue**:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Data Types

In Python, a data type defines what kind of value a variable is storing.
It helps Python decide what operations are allowed on the value.

Primitive Data Types (Basic / Built-in) :

These are the simple, basic types. They store single values and are already built into Python.

Data Type	Example	Description
int	10 , -5	Whole Number
float	3.14 , 2.0	Decimal numbers
bool	True , False	Logical Values
str	"hello"	Text(string)
NoneType	None	Represents "Nothing" value

Non-Primitive Data Types (Complex / Collection Types) :

These are not basic, but used to store multiple values or custom structures.

Date Type	Example	Description
list	[1,2,3]	Changeable, ordered collection
tuple	(4,5,6)	Fixed , ordered collection
set	{1,2,3}	Unique , unordered Values
dict	{'name' : 'Ashish'}	Key-value pairs
range	range(4)	Sequence of Numbers
bytes	b'abc'	Immutable byte data
bytearray	bytearray(4)	Mutable binary data

Key Difference:

Primitive	**Non Primitive**
Single Simple Value	Group/structure of data
Built in & small	Can be large/ Complex
Cannot be customized	Can hold multiple types

List in python:

A list is a versatile collection in Python that holds multiple values in one variable.
Lists are:

- **Ordered** – Items are stored in a defined sequence
- **Mutable** – You can change their contents
- **Duplicate-friendly** – Repeating items are allowed

Creating a List

```
list1 = [1, 2, 3, 4, 5]              # Integer list
list2 = ["apple", "banana", "mango"]  # String list
list3 = [10, "apple", 3.14, True]     # Mixed data types
empty_list = []                       # Empty list
```

Accessing List Elements

Indexing

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])    # First element → 10
print(my_list[-1])   # Last element → 50
```

Slicing

```
print(my_list[0:2])   # [10,20]
```

Operation	Syntax / Code	Description
Length	len(list1)	Number of items in the list
Append	list1.append(6)	Adds item at the end
Insert	list1.insert(2, 99)	Inserts 99 at index 2
Remove	list1.remove(3)	Removes first occurrence of 3
Pop	list1.pop() or list1.pop(2)	Removes and returns last/item at index
Index	list1.index(4)	Returns index of first occurrence of 4
Count	list1.count(2)	Counts how many times 2 appears
Reverse	list1.reverse()	Reverses the list
Sort	list1.sort()	Sorts in ascending order
Copy	copy_list = list1.copy()	Creates a copy
Clear	list1.clear()	Empties the list

Combining Lists

```
a = [1, 2, 3]
b = [4, 5]
c = a + b         # [1, 2, 3, 4, 5]
a.extend(b)       # a becomes [1, 2, 3, 4, 5]
```

Check Existence

```
if "banana" in list2:
    print("Found!")
```

Looping through a List

```
for item in list2:
    print(item)
```

List Comprehension (Shorter Syntax)

```
squares = [x*x for x in range(1, 6)]
print(squares)   # [1, 4, 9, 16, 25]
```

List vs Array in Python

- Lists can store different types.
- Arrays (from array module) store elements of the same type and are more memory-efficient

Tuple in python:

In Python, a **tuple** is an **ordered, immutable** (unchangeable) collection of items. Tuples are used to group related data together and save memory.

Creating Tuples

```
t1 = (1, 2, 3)
t2 = ("apple", "banana", "cherry")
t3 = (1, "hello", 3.14)
t4 = ()              # Empty tuple
t5 = (5,)             # Single-element tuple (comma is necessary)
```

Note: Use commas to define a tuple, even for one item.

Accessing Tuple Elements

```
t = ("a", "b", "c", "d")
print(t[0])     # a
print(t[-1])    # d
print(t[1:3])   # ('b', 'c')
```

Tuples Are Immutable

```
t = (10, 20, 30)
# t[0] = 100      ❌Error: 'tuple' object does not support item assignment
```

Operation	**Example**	**Result / Use**
len()	len(t1)	Number of elements
count(x)	t1.count(2)	Count of element 2
index(x)	t1.index(3)	First index of element 3
Concatenation	t1 + t2	Joins two tuples
Repetition	t1 * 2	Duplicates tuple
Membership	2 in t1	Checks if 2 is in tuple

Looping Through a Tuple

```
for item in t2:
   print(item)
```

Tuple Packing and Unpacking

```
# Packing
person = ("Ashish", 22, "India")
# Unpacking
name, age, country = person
print(name)     # Ashish
print(age)     # 22
print(country)  # India
```

When to Use Tuples

- When the data **should not change**
- For **function returns** of multiple values
- To **optimize memory and speed**

Dictionary in python:

A **dictionary** in Python is an **unordered**, **mutable**, and **indexed** collection of key-value pairs. It's one of the most powerful built-in data types for fast lookups.

Creating a Dictionary

```
# Using curly braces
my_dict = {"name": "Ashish", "age": 22, "city": "Bhiwani"}
# Using dict() constructor
my_dict2 = dict(name="John", age=30)
# Empty dictionary
empty_dict = {}
```

Accessing Elements

```
print(my_dict["name"])      # Ashish
print(my_dict.get("age"))   # 22
```

- ✓ **get()** returns **None** if the key doesn't exist (safe).
- ❌ Using **[]** with a missing key will raise an error.

Adding & Updating Items

```
my_dict["email"] = "ashish@example.com"  # Add new key-value
my_dict["age"] = 23                     # Update value
```

Deleting Items

```
del my_dict["city"]        # Deletes key 'city'
my_dict.pop("age")         # Removes and returns value
my_dict.clear()            # Empties the dictionary
```

Looping through Dictionary

```
for key in my_dict:
   print(key, my_dict[key])
# OR
for key, value in my_dict.items():
   print(key, "→", value)
```

Useful Dictionary Methods

Method	Example	Description
keys()	my_dict.keys()	Returns all keys
values()	my_dict.values()	Returns all values
items()	my_dict.items()	Returns all key-value pairs
get(key)	my_dict.get("name")	Returns value of key
update()	my_dict.update({"age": 25})	Updates or adds
pop(key)	my_dict.pop("city")	Removes and returns value of key
clear()	my_dict.clear()	Empties the dictionary

Example: Student Record

```
student = {
   "name": "Ashish",
   "roll": 101,
   "marks": {"Math": 90, "Sci": 95}
}
print(student["marks"]["Math"])  # Access nested value → 90
```

Why Use Dictionaries?

- Fast **lookup** (faster than lists/tuples)
- Useful for **structured data**
- Key-value format is ideal for **real-world objects**

Set in python:

A **set** in Python is an **unordered**, **mutable**, and **unindexed** collection that **does not allow duplicate elements**. It's mainly used for **membership testing**, **removing duplicates**, and performing **mathematical set operations**.

Creating a Set

```
s1 = {1, 2, 3, 4}
s2 = set(["apple", "banana", "apple", "mango"])  # duplicates removed
empty_set = set()  # ❗ Not {} — that creates an empty dictionary
```

Sets can only contain **immutable (hashable)** elements (no lists/dicts inside).

Accessing Set Elements

- Sets are **unordered**, so they **cannot be accessed using an index**.
- Use a loop to access elements:

```
for item in s1:
  print(item)
```

Adding Elements

```
s1.add(5)            # Add single element
s1.update([6, 7, 8])    # Add multiple elements
```

Removing Elements

```
s1.remove(2)    # Removes 2 — raises error if not found
s1.discard(3)   # Removes 3 — no error if not found
s1.pop()        # Removes random element
s1.clear()      # Empties the set
```

Set Operations

Operation	**Syntax**	**Result**
Union	`s1	s2orss1.union(s2)`
Intersection	s1 & s2 or s1.intersection(s2)	Common elements
Difference	s1 - s2 or s1.difference(s2)	Items in s1 but not in s2
Symmetric Diff	s1 ^ s2 or s1.symmetric_difference(s2)	Items in either, but not both
Subset	s1.issubset(s2)	True if s1 is inside s2
Superset	s1.issuperset(s2)	True if s1 contains s2
Disjoint	s1.isdisjoint(s2)	True if no common elements

Set Example: Removing Duplicates

```
nums = [1, 2, 2, 3, 4, 4, 5]
unique_nums = set(nums)
print(unique_nums)  # {1, 2, 3, 4, 5}
```

Set vs List vs Tuple vs Dictionary

Feature	**List ([])**	**Tuple (())**	**Set ({})**	**Dictionary ({k: v})**
Ordered	Yes	Yes	No	Yes (3.7+)
Duplicates	Allowed	Allowed	Not Allowed	Keys must be unique
Mutable	Yes	No	Yes	Yes
Indexed	Yes	Yes	No	Keys only

String in python :

Strings in Python are sequences of characters and support a wide variety of operations.

Creating Strings

```
s1 = "Hello"
s2 = 'World'
s3 = """Multiline
String"""
```

Accessing & Slicing

```
s = "Python"
print(s[0])      # 'P'
print(s[-1])     # 'n'
print(s[1:4])    # 'yth'
print(s[::-1])   # Reverse → 'nohtyP'
```

String Built-in Methods

Method	Example	Description
lower()	"HELLO".lower()	'hello'
upper()	"hello".upper()	'HELLO'
title()	"hello world".title()	'Hello World'
capitalize()	"python".capitalize()	'Python'
strip()	" hello ".strip()	'hello' (removes leading/trailing spaces)
lstrip() / rstrip()	" hi ".lstrip() / .rstrip()	Removes left/right spaces
replace(a, b)	"apple".replace("a", "A")	'Apple'
count(x)	"banana".count("a")	3
find(x)	"banana".find("a")	1 (first occurrence)
index(x)	"banana".index("n")	2 (like find() but gives error if not found
startswith("P")	"Python".startswith("P")	True
endswith("n")	"Python".endswith("n")	True
split()	"a b c".split()	['a', 'b', 'c'] (default is space)
join(list)	' '.join(['I','am','GPT'])	'I am GPT'
isalpha()	"abc".isalpha()	True (letters only)
isdigit()	"123".isdigit()	True (numbers only)

Concatenation & Repetition

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

Membership Testing

```
"Py" in "Python"        # True
"java" not in "Python"  # True
```

Looping through String

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

Escape Sequences

Escape Code	Meaning
\n	Newline
\t	Tab
\\	Backslash
\"	Double quote
\'	Single quote

Immutability of Strings

```
  s = "hello"
  # s[0] = "H" ❌ Error — strings are immutable
✓ s = "H" + s[1:]  #  Create a new string
  print(s)        # Hello
```

String Formatting

```
name = "Ashish"
age = 22
print(f"My name is {name} and I am {age} years old.")    # f-string
print("Name: {}, Age: {}".format(name, age))          # format()
```

Truthy and Falsy Values in Python :

In Python, every value can be used in conditions like if, and Python automatically treats it as either:

- Truthy → Treated as True
- Falsy → Treated as False

Even if the value is not True or False, Python decides based on its “truthiness”.

Falsy Values

The following are considered false when checked in conditions:

Value	Reason
None	Represents “nothing”
False	Boolean false
0 , 0.0	Numeric zero
""	Empty string
[]	Empty List
{}	Empty Dictionary
()	Empty Tuple
Set()	Empty set

Example:

```
 if "":
     print("This is truthy")
else:
    print("This is falsy")
 # Output: This is falsy
```

Truthy Values

Anything that’s not falsy is treated as truthy:

Example	Description
"text"	Non-Empty String
123 , 3.14	Non-Zero numbers
[1]	Non-Empty list
{"x":1}	Non-Empty Dictionary
True	Boolean True

Example:

```
 if [1, 2]:
     print("Truthy list")

# This will print
```

and **and** or **Behavior with Truthy/Falsy :**

Python’s and/or return actual values based on truthiness.

and Operator	**Or Operator**
• Returns the **first falsy** value it finds • If all are truthy, returns the **last one** Examples: print(0 and 10) # 0 (first falsy) print("hi" and "there") # "there" (last truthy)	• Returns the first truthy value • If all are falsy, gives the last one Examples: print("" or "hello") #"hello" print(0 or "" or None) #None (all falsy)

Expression	5 and 10	0 and 7	"" or "yes"	0 or "" or None
Output	10	0	"yes"	None
Why	Both truthy --> return last	First Falsy return	First Truthy returned	All are falsy --> return last

Functions

A **function** is a block of organized, reusable code used to perform a single, related action.

Functions make the code **modular**, **reusable**, **readable**, and **easy to debug**.

- Avoid repeating code
- Keep programs organized
- Make code reusable and readable

Types of Functions

Built-in Functions – Already available in Python

Examples: print(), len(), range(), sum(), type(), input()

User-defined Functions – Created by users using the def keyword

Defining a Function

```
def function_name(parameters):
   """docstring (optional): describes what the function does"""
   # block of code
   return result  # (optional)
```

Example:

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

Calling a Function

You call or invoke the function by writing its name followed by parentheses:

greet("Ashish")

Return Statement

The return statement is used to return a value from the function.

```
def add(a, b):
   return a + b
result = add(5, 3)
print(result)  # Output: 8
```

Function Parameters & Arguments

1. Positional Arguments

```
def multiply(a, b):
   return a * b
print(multiply(3, 4))  # Output: 12
```

2. Keyword Arguments

```
def divide(a, b):
   return a / b
print(divide(b=10, a=100))  # Output: 10.0
```

3. Default Arguments

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

4. Variable-Length Arguments

***args (non-keyworded variable-length arguments – tuple)**

```
def sum_all(*numbers):
   return sum(numbers)
print(sum_all(1, 2, 3, 4))  # Output: 10
```

****kwargs (keyworded variable-length arguments – dictionary)**

```
def display_info(**info):
   for key, value in info.items():
      print(f"{key}: {value}")
display_info(name="Ashish", age=25)
```

Lexical Scope:

Lexical scope means that the scope of a variable is determined by its **position in the source code** (not at runtime).
In Python, **functions remember the scope in which they were defined**, not the scope from which they were called.

Example:

```
def outer():
   x = 10  # Enclosed scope
def inner():
      print(x)  # Looks for x in enclosing scope
   inner()
outer()
```

Output: 10

The inner() function uses x from outer() — this is **lexical scope**.

Scope Levels in Python (LEGB Rule):

Scope	Description
Local	Inside the current function
Enclosing	Inside any enclosing functions (nested)
Global	At the top level of the script/module
Built-in	Predefined names in Python (len, print)

LEGB in Action:

```
x = "global"

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

Output: local

If **local** was not defined:

```
def inner():
   print(x)
```

Then **x = "enclosing"** would be used.

First-Class Functions

In Python, **functions are first-class objects**, meaning:

- They can be **assigned to variables**
- **Passed as arguments**
- **Returned from other functions**
- **Store them in collections like lists or dictionaries**

Example1: Assigning Function to a Variable

```
def greet(name):
   return f"Hello, {name}"
say_hello = greet  # assigned to variable
print(say_hello("Ashish"))  # Output: Hello, Ashish
```

Example2:Passing a Function as Argument

```
def apply_twice(func, x):
         return func(func(x))
def add_three(n):
          return n + 3
 print(apply_twice(add_three, 5)) # Output: 11
```

Example3:Returning a Function

```
def outer():
    def inner():
        return "Inside inner function!"
    return inner            # return the function, not the result
my_func = outer()
print(my_func())            # Output: Inside inner function!
```

Example4:Storing Functions in a List

```
def square(n):
    return n * n
def cube(n):
    return n ** 3
operations = [square, cube]

for func in operations:
    print(func(2))          # Output: 4, 8
```

Higher-Order Functions

A **Higher-Order Function** is a function that:

Takes one or more functions as arguments
OR returns a function as its result
(or both)

In Python, functions are **first-class objects**, so you can:

- Pass them as arguments
- Store them in variables
- Return them from other functions

1. HOF that Takes a Function as an Argument

Example: apply_twice(func, value)

```
def square(x):
  return x * x
def apply_twice(func, value):
  return func(func(value))
print(apply_twice(square, 2))  # Output: 16
```

Explanation:

- square(2) → 4
- Then again: square(4) → 16

2. HOF that Returns Another Function

Example: make_multiplier(n)

```
def make_multiplier(n):
  def multiplier(x):
    return x * n
  return multiplier
times3 = make_multiplier(3)
print(times3(5))  # Output: 15
```

Explanation:

- make_multiplier(3) returns a function multiplier that multiplies by 3.

3. HOFs in Real Python: map(), filter(), reduce()

Example: map() (Applies a function to every item)

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x*x, nums))
print(squared)  # Output: [1, 4, 9, 16]
```

Example: filter() (Filters items based on a condition)

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

Example: reduce() (Reduces the list to a single value)

```
from functools import reduce

nums = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, nums)
print(product)  # Output: 24
```

4. HOF in Sorting: sorted() with custom key

```
names = ["ashish", "sheoran", "gpt", "openai"]
sorted_names = sorted(names, key=lambda x: len(x))
print(sorted_names)  # Output: ['gpt', 'ashish', 'openai', 'sheoran']
```

5. HOF with Decorators (Advanced Use)

```
def uppercase_decorator(function):
    def wrapper():
        result = function()
        return result.upper()
    return wrapper
@uppercase_decorator
def say_hello():
    return "hello world"
print(say_hello())  # Output: HELLO WORLD
```

Summary

Example Type	What It Does
apply_twice(func, value)	Takes function as argument
make_multiplier(n)	Returns a new function
map(func, iterable)	Applies function to all items
filter(func, iterable)	Filters items using a function
reduce(func, iterable)	Reduces to single value
@decorator	Adds extra behavior to a function

Why Use Higher-Order Functions?

- Helps in **writing clean, concise, and reusable code**
- Makes **functional programming** possible in Python
- Often used in data processing, UI behavior, mathematical modeling, etc.

Lambda Function:

A **lambda function** in Python is a **small, anonymous function** defined using the lambda keyword instead of def.

It's typically used when you need a simple function for a short time — usually **one-liner logic**.

Syntax:

```
lambda arguments: expression
```

- No def or return keyword needed.
- The **expression must be a single line** and will be automatically returned.

Example:

```
add = lambda a, b: a + b
print(add(3, 4))  # Output: 7
```

Used in functions like map(), filter(), sorted():

```
nums = [5, 2, 9]
sorted_nums = sorted(nums, key=lambda x: -x)  # Sort descending
print(sorted_nums)  # Output: [9, 5, 2]
```

Lambda with Conditional Expression

```
max_func = lambda a, b: a if a > b else b
print(max_func(5, 8))  # Output: 8
```

When NOT to Use Lambda

- When function logic is complex (use def)
- When debugging — lambdas have no name or docstring

Summary

- Lambda = one-liner function: lambda args: expression
- Often used with map, filter, reduce, sorted
- Cannot contain multiple statements
- Great for short, temporary functions

Decorator Functions:

A **decorator** is a **function that modifies the behavior of another function** without changing its source code.

Decorators are a key feature in **Python's functional programming**, often used for:

- Logging
- Timing
- Access control
- Memoization (caching)
- Web frameworks (like Flask, Django)

Real-World Analogy:

Think of a decorator like a gift wrapper. The core gift (function) stays the same, but the wrapper (decorator) adds something extra.

Basic Structure of a Decorator

```
def decorator_function(original_function):
    def wrapper_function():
        print("Before the function runs")
        original_function()
        print("After the function runs")
    return wrapper_function
```

Applying a Decorator

Without @ syntax:

```
def say_hello():
    print("Hello!")
decorated = decorator_function(say_hello)
decorated()
```

With @ decorator syntax (cleaner):

```
@decorator_function
def say_hello():
    print("Hello!")
say_hello()
```

Example: Logging Decorator

```
def logger(func):
    def wrapper():
        print(f"Calling function: {func.__name__}")
        func()
        print(f"Finished calling: {func.__name__}")
    return wrapper
@logger
def greet():
    print("Hi!")
greet()
```

Output:

```
Calling function: greet
Hi!
Finished calling: greet
```

Decorator with Arguments

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator
@repeat(3)
def say_hi():
    print("Hi!")
say_hi()
```

Output:

```
Hi!
Hi!
Hi!
```

Built-in Decorators

Decorator	**Use**
@staticmethod	Inside class, no access to self
@classmethod	Takes cls instead of self
@property	Turns a method into a property

Modules :

A **module** is a single Python file (.py) that contains Python code – like functions, classes, or variables – that you can reuse in other programs.

Example:

```
# greetings.py (this is a module)
def say_hello(name):
   print(f"Hello, {name}!")
```

You can import and use this module:

```
import greetings
greetings.say_hello("Ashish")       # Output: Hello, Ashish!
```

How to Create a Module:

Create a .py file with some reusable code.

Import it in another script using import filename

Package :

A **package** is a directory (folder) that contains multiple **modules** and an optional special file named __init__.py.

- The __init__.py file tells Python that the folder is a package.
- It can be empty or contain initialization code for the package.

Structure Example:

```
my_package/
 |
 ├── __init__.py
 ├── greetings.py
 └── math_tools.py
```

Type	Example
Built-in	math, os, random
Third-party	numpy, pandas, matplotlib (installed via pip)
User-defined	Your own .py files

Using the package:

```
# Inside greetings.py
def say_hi():
   print("Hi!")
# Inside another script
from my_package import greetings
greetings.say_hi()
```

Import Types

Syntax	Description
import module_name	Imports the whole module
from module_name import function_name	Imports specific function
from package import module	Imports module from package
import module as alias	Gives the module a short name

Concept	Module	Package	LIbrary
Structure	Single .py file	A folder with __init__.py	One or more packages/modules
Contains	Functions, classes, variables	Multiple modules and subfolders	Ready-to-use code collectiton
Example	math.py	Mypackage/	numpy, scikit-learn, pandas

Package Manager:

In Python, **package managers** help you install, update, and manage external libraries and packages.
The most commonly used package managers are: **PIP , Conda , UV**

1. pip – Python’s Default Package Manager

- Comes pre-installed with Python (version 3.4+).
- Used to install packages from [PyPI (Python Package Index)](https://pypi.org).

Common pip Commands:

```
pip install package_name          # Install a package
pip install numpy==1.22.0         # Install specific version
pip install -r requirements.txt   # Install from file
pip list                          # List installed packages
pip show package_name             # Show details of package
pip uninstall package_name        # Remove a package
pip freeze > requirements.txt     # Export current environment
```

2. conda – Package + Environment Manager

- Used mostly with **Anaconda/Miniconda** distributions.
- Can install both Python and non-Python packages (like C/C++ libs).

Commands:

```
conda install package_name         # Install from conda repositories
conda list                         # List installed packages
conda update package_name          # Update a package
conda remove package_name          # Remove a package
conda create --name env_name       # Create new environment
conda activate env_name            # Activate environment
```

3. uv – A Fast Python Package Manager

uv is a **modern, ultra-fast** Python package manager created by **Astral** . It is **significantly faster** than **pip**, **pip-tools,** and even **poetry** for dependency resolution and installation.

Why Use uv?

Feature	Description
Speed	Super fast (written in Rust)
Compatibility	Drop-in replacement for pip and pip-tools
Lock files	Generates requirements.txt, requirements.lock, uv.lock
No .venv by default	Can work with or without virtual environments
Reproducible installs	Like pip-tools, but faster

Common uv Commands:

Task	uv Command
Install package	uv pip install <package>
Use requirements.txt	uv pip install -r requirements.txt
Freeze installed packages	uv pip freeze
Create lockfile	uv pip compile
Update packages	uv pip compile --upgrade
Run scripts	uv venv exec python script.py

It mimics pip, so most pip commands also work with uv pip.

What is Docker?

Docker is a containerization platform that packages your application code and its dependencies (libraries, environment settings, etc.) into an isolated unit called a container.

Think of it as a lightweight virtual machine, but faster and more efficient.

Why Use Docker with Python?

Consistent Environments – No more "works on my machine" problems.
Dependency Isolation – Python packages won't interfere with each other.
Easy Deployment – One command can run the same code on any system.
Scalability – Works well in microservices and cloud environments.

Common Use Cases for Docker in Python

Use Case	Description
Web Apps	Package Flask or Django apps with their dependencies.
Data Science	Share Jupyter notebooks with all required libraries.
APIs	Deploy FastAPI/Flask-based microservices easily.
Testing	Spin up isolated test environments with Docker Compose.
CI/CD	Integrate Docker in pipelines for building/testing/deploying code.

Dependency Resolution:-

Dependency resolution in Python refers to the process of identifying, installing, and managing the correct versions of external **libraries (dependencies)** that your Python project needs to run properly.

What Are Dependencies?

Dependencies are external packages or modules your Python code relies on. For example:

```
import requests  # 'requests' is an external dependency
```

Dependency Resolution Process

When you install dependencies (like with **pip**), Python tools try to:

Read your project's requirements

- □ From files like **requirements.txt**, **pyproject.toml**, or **Pipfile**.

Find compatible versions

- □ If you need **packageA==1.2** and **packageB** needs **packageA>=1.3**, this creates a **conflict**.
- □ Tools try to find versions that satisfy all constraints.

Download and install

- □ Once compatible versions are resolved, they are downloaded from **PyPI** and installed.

Tools That Do Dependency Resolution

Tool	Description
pip	Default installer and resolver (since v20.3, pip has a new resolver that handles conflicts better).
pip-tools	Better handling of pinned versions (pip-compile).
poetry	Advanced dependency resolver with pyproject.toml.
conda	Used in scientific environments; resolves packages and environments.

Example

Suppose you have this requirements.txt:

```
flask==2.2.0
requests>=2.25
```

When you run:

```
pip install -r requirements.txt
```

- □ Pip checks compatibility between Flask 2.2.0 and requests>=2.25.
- □ If both can coexist, it installs them.
- □ If not, it throws a **dependency conflict error**.

Example of Conflict

You try to install:

```
pip install packageA packageB
```

But:

- packageA requires numpy==1.19
- packageB requires numpy>=1.21

Then pip cannot resolve the conflict and throws an error like:

```
ERROR: Cannot install packageA and packageB because of conflicting dependencies.
```

How to Check Dependencies

```
pip show <package_name>
pipdeptree                # to see dependency tree
```

CAP Theorem (Brewer's Theorem)

The **CAP theorem** is a fundamental principle in distributed systems that states:

A distributed system can **only guarantee two** of the following three properties at the same time:

Property	Meaning
C - Consistency	Every read gets the most recent write (no stale data).
A - Availability	Every request gets a (non-error) response — even if it's not the latest data.
P - Partition Tolerance	The system continues to operate even if there's a network failure (partition) between nodes.

Examples of CAP Combinations

Type	Description	Example Systems
CP (Consistency + Partition Tolerance)	System is consistent and partition-tolerant but may sacrifice availability (might refuse requests during failure).	HBase, MongoDB (in some configs), Zookeeper
CA (Consistency + Availability)*	Works only when there's no partition; not realistic in distributed systems.	Traditional relational databases (on a single server)
AP (Availability + Partition Tolerance)	System is available and partition-tolerant, but might return stale data (eventual consistency).	Cassandra, DynamoDB, CouchDB

***CA is only possible when there's no partition — hence in real-world distributed systems, P (Partition Tolerance) is a must, and systems choose between C and A.**

Property	Guarantees
Consistency	All nodes see the same data at the same time
Availability	Every request gets a response
Partition Tolerance	System works despite network failure

Real-World Tradeoff Example

Imagine a messaging app:

- If you choose **CP**: A message won't show up until all servers agree (strong consistency), but during a network failure, users might see an error.
- If you choose **AP**: Messages always show (availability), even during network issues, but might be out-of-order (eventual consistency).

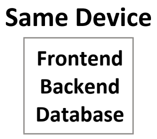
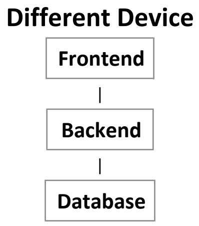
CAP Triangle

Imagine a triangle:

You can **only pick two** of the three:

```
        Consistency
            /\
           /  \
          /    \
         /      \
        /        \
  Availability   Partition Tolerance
```

Monolithic vs. Distributed Systems

<table>
<tr><th>Monolithic Architecture</th><th>Distributed Architecture</th></tr>
<tr><td>A <b>monolithic system</b> is a <b>single, unified unit</b>. All the components (UI, business logic, database access) are packaged together and run as a single service</td><td>A <b>distributed system</b> is one where <b>components are split across multiple machines/services</b>, but work together as one system.</td></tr>
<tr><td>
<table>
<tr><th>Characteristics</th><th>Drawbacks</th></tr>
<tr><td>• All functionality is deployed together.
• Simple to develop and test initially.
• Easy to deploy as one unit.</td><td>• Hard to scale specific parts.
• A small bug can crash the whole system.
• Deployment becomes slow with growing size.
• Difficult to adopt new technologies per module.</td></tr>
</table>
</td><td>
<table>
<tr><th>Characteristics</th><th>Drawbacks</th></tr>
<tr><td>• Loosely coupled components (often via APIs).
• Easier to scale individual parts.
• Better fault tolerance.
• Teams can work on different services independently.</td><td>• More complex (networking, failure handling).
• Requires service discovery, load balancing, monitoring, etc.
• Debugging and testing are harder.</td></tr>
</table>
</td></tr>
<tr><td><b>Example:</b>
A traditional e-commerce web app where:
• Product management
• Order processing
• Payment
• Authentication are all in <b>one codebase and deployed together</b>.</td><td><b>Example Architectures:</b>
• <b>Microservices</b>: Each component is its own service (e.g., user-service, order-service).
• <b>Distributed databases</b>: Data spread across nodes (e.g., Cassandra, MongoDB cluster).
• <b>Cloud-native apps</b> using Docker, Kubernetes, etc.</td></tr>
<tr><td>
Same Device
Frontend
Backend
Database</td><td>
Different Device
Frontend
Backend
Database</td></tr>
</table>

Feature	Monolithic	Distributed (Microservices)
Structure	Single codebase & deployment	Multiple independent services
Scalability	Vertical (scale whole app)	Horizontal (scale specific services)
Deployment	All at once	Independent deployment
Technology Stack	Same for all parts	Can vary per service
Failure Impact	One crash = whole app down	Only the failing service is down
Performance	Fast communication (in-process)	Slower (network latency)
Complexity	Low	High (requires DevOps, coordination)

Exception Handling

Exception Handling allows you to handle runtime errors, so your program doesn't crash when unexpected events occur (like dividing by zero, missing files, etc.)

Why Use Exception Handling?

Without handling:

```
print(10 / 0)          # ZeroDivisionError: division by zero
```

With handling:

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Basic Syntax:

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Code to run if exception occurs
else:
    # Code to run if no exception occurs (optional)
finally:
    # Code that runs no matter what (optional)
```

Example with All Clauses:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Please enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Result is:", result)
finally:
    print("This always runs.")
```

Common Exceptions:

Exception	Description
ZeroDivisionError	Division by zero
ValueError	Wrong value (e.g., invalid int)
TypeError	Wrong data type
FileNotFoundError	File not found
IndexError	List index out of range
KeyError	Missing key in a dictionary
ImportError	Module not found

Raising Exceptions Manually:

```
raise ValueError("This is a custom error message.")
```

Custom Exception Class:

```
class MyError(Exception):
    pass
try:
    raise MyError("Something went wrong")
except MyError as e:
    print("Caught:", e)
```

File Handling :-

File handling in Python allows you to **create**, **read**, **write**, and **delete** files. Python provides built-in functions to work with files, which are commonly used for storing data permanently.

Basic Syntax

```
file = open("filename.txt", "mode")
# do operations
file.close()
```

File Modes

Mode	'r'	'w'	'a'	'x'	'b'	't'	'+'
Description	Read (default). Error if file not found.	Write. Creates file or overwrites.	Append. Creates file or adds to end	Create. Error if file exists	Binary mode (e.g., 'rb', 'wb')	Text mode (default)	Read and write

Reading a File	Writing to a File	File Checking

with open("sample.txt", "r"**) as** f:
    print(f.**read()**)   # Reads entire file
    # f.**readline()**   # Reads one line
    # f.**readlines()**   # Reads all lines into a list
 	
with open("data.txt", "r"**) as** f:
    for line **in** f:
        print(line.**strip()**)
	with open("sample.txt", "w"**) as** f:
	    f.**write**("Hello, world!")
	
	□ w overwrites the file.
	□ Use a to append
	
	with open("sample.txt", "a"**) as** f:
	    f.**write**("\nAppended text.")
		import os
		if os.path.exists("sample.txt"**)**:
		    print("File exists.")
		else:
		    print("File does not exist.")
		
		To **delete a file**:
		        os.remove("sample.txt"**)**

Why Use with open(...) as f:?

- Automatically **closes** the file after block ends   (Context Manager **with**)
- Prevents file corruption or memory leaks

What is Jupyter Lab?

JupyterLab is an advanced version of the classic **Jupyter Notebook** interface. It provides an **IDE-like** environment for working with:

- Python code
- Data (CSV, Excel, JSON, etc.)
- Terminals and Consoles
- Plots (Matplotlib, Plotly)
- Extensions (e.g., Git, Table of Contents)

Key Features of JupyterLab

Feature	Description
Multiple Tabs	Open notebooks, terminals, text files, and more side by side
Drag & Drop	Move and arrange files in the interface
Extensions	Add Git, variable inspector, debugger, etc.
Interactive Output	Supports widgets, plots, and real-time display
Integrated Terminal	Run shell commands directly in JupyterLab

How to Install JupyterLab		How to Run JupyterLab
Using pip	**Using conda**	
pip install jupyterlab	conda install -c conda-forge jupyterlab	**jupyter lab**

This will:

- Launch a local web server
- Open JupyterLab in your browser (http://localhost:8888)

Object-Oriented Programming

Object-Oriented Programming (OOP) in **Python** is a programming paradigm based on the concept of “**objects**,” which can contain data (**attributes**) and code (**methods**). Python supports all key OOP principles:

Four Pillars of OOP

Encapsulation
Abstraction
Inheritance
Polymorphism

Class and Object

- **Class**: A blueprint for creating objects.
- **Object**: An instance of a class.

```
class Person:
  def __init__(self, name, age):
    self.name = name      # Attribute
    self.age = age
def greet(self):              # Method
    print(f"Hello, my name is {self.name}.")
p1 = Person("Alice", 30)  # Object
p1.greet()
```

Encapsulation

Wrapping data (**attributes**) and methods inside a single unit (**class**) and restricting direct access.

Goals of Encapsulation:	**Why Use Encapsulation?**	**Real Life Analogy**
Protect data from being modified directly. **Control how data is accessed/modified** using methods. **Hide implementation details** (data hiding).	Prevent misuse of sensitive data. Create getter/setter methods to **validate data**. Make code **modular** and **secure**.	**Think of a TV remote:** **You can use buttons (methods) to change the channel.** **But you cannot access internal circuits (private data) directly.**

Example:

```
class BankAccount:
  def __init__(self, balance):
    self.__balance = balance              # private variable
def deposit(self, amount):
    self.__balance += amount
def get_balance(self):
    return self.__balance
acc = BankAccount(1000)
acc.deposit(500)
print(acc.get_balance())                 # Output: 1500
# print(acc.__balance)                   # Error: private variable
```

```
class Student:
    def __init__(self, name, marks):
        self.name = name         # public attribute
        self.__marks = marks     # private attribute

    def set_marks(self, marks):
        if 0 <= marks <= 100:
            self.__marks = marks
        else:
            print("Invalid marks")

    def get_marks(self):
        return self.__marks

# Creating object
s1 = Student("Ashish", 85)

print(s1.name)                #  Accessible
print(s1.get_marks())         # Access via method

s1.set_marks(95)              #  Safe modification
print(s1.get_marks())

# print(s1.__marks)           ❌ Error: Private attribute not directly accessible
```

Access Modifiers in Python

Access Modifier	Syntax	Access Scope
Public	self.name	Anywhere
Protected	self._name	Convention: Meant for internal use or subclass
Private	self.__name	Name mangling: Can't be accessed directly

```
  class Test:
     def __init__(self):
        self.public = "I am public"
        self._protected = "I am protected"
        self.__private = "I am private"
  t = Test()
✓ print(t.public)
  print(t._protected)         # Avoid, but possible
  # print(t.__private)         # ❌ Error
✓ print(t._Test__private)     # Name-mangled access
```

Abstraction

Hides the internal implementation and only shows the necessary details.
Focuses on **what an object does**, not **how it does it**.

Purpose of Abstraction	**In Python, Abstraction is done using:**	**Rules of Abstraction:**	**Real Life Analogy**
Hide complexity Make code cleaner and more secure Enforce a structure for subclasses	**Abstract Base Classes (ABC)** @abstractmethod decorator from the abc module	Any class with at least one @abstractmethod becomes **abstract**. You **cannot create an object** of an abstract class. Subclasses must **override all abstract methods** to be instantiable. You can have **concrete methods** in an abstract class too.	Think of a **Bank ATM**: You interact with buttons like *withdraw, check balance* (interface). You don't see **how the backend code connects to the server** or database.

Example using ABC:

```
from abc import ABC, abstractmethod
class Animal(ABC):
   def eat(self):          # Concrete method
    print("This animal eats food.")
  @abstractmethod
  def make_sound(self):    # Abstract method
    pass
class Dog(Animal):
  def make_sound(self):
    print("Woof!")
d = Dog()
d.eat()
d.make_sound()
```

```
from abc import ABC, abstractmethod

# Abstract Base Class
class Vehicle(ABC):

  @abstractmethod
  def start_engine(self):
    pass

  @abstractmethod
  def stop_engine(self):
    pass

# Concrete Subclass
class Car(Vehicle):
  def start_engine(self):
    print("Car engine started.")

  def stop_engine(self):
    print("Car engine stopped.")

# Cannot instantiate abstract class directly:
# v = Vehicle()  ❌ TypeError

# Create object of subclass
c = Car()
c.start_engine()  #  Output: Car engine started.
c.stop_engine()   # Output: Car engine stopped.
```

Inheritance

Allows one class (child) to inherit attributes, properties and methods from another (parent).
It helps in **code reuse**, **extensibility**, and representing **real-world hierarchies**.

Example:

```
class Vehicle:
   def __init__(self, brand):
      self.brand = brand
def start(self):
      print(f"{self.brand} started.")
class Car(Vehicle):                  # Child class
   def drive(self):
      print(f"{self.brand} is driving.")
c = Car("Toyota")
c.start()
c.drive()
```

Why Use Inheritance?

Avoid code duplication
Enable **"is-a" relationships**
Build on existing classes

Types of Inheritance in Python

Type	Description
Single	One child, one parent
Multiple	One child, multiple parents
Multilevel	Chain of inheritance (grandparent → parent → child)
Hierarchical	One parent, multiple children
Hybrid	Combination of multiple types

Real Life Analogy

A **Car** class inherits from a **Vehicle** class:
All cars are vehicles (is-a relationship)
Inherits methods like .start(), .stop()

Single Inheritance

```
class Animal:
   def speak(self):
      print("Animal speaks")
class Dog(Animal):
   def bark(self):
      print("Dog barks")
d = Dog()
d.speak()
d.bark()
```

Multilevel Inheritance

```
class Grandparent:
   def family_name(self):
      print("Sheoran")
class Parent(Grandparent):
   def occupation(self):
      print("Engineer")
class Child(Parent):
   def hobby(self):
      print("Painting")
c = Child()
c.family_name()
c.occupation()
c.hobby()
```

Multiple Inheritance

```
class Father:
   def skill(self):
print("Carpenter")
class Mother:
   def hobby(self):
      print("Dancer")
class Child(Father, Mother):
   pass
c = Child()
c.skill()
c.hobby()
```

Hierarchical Inheritance

```
class Vehicle:
   def fuel(self):
      print("Fuel used")
class Bike(Vehicle):
   def wheels(self):
      print("2 wheels")
class Car(Vehicle):
   def doors(self):
      print("4 doors")
b = Bike()
b.fuel()
b.wheels()
c = Car()
c.fuel()
c.doors()
```

Hybrid Inheritance

(Combination of multiple types — handled with **Method Resolution Order** or MRO)

```
class A:
   def show(self):
      print("A")
class B(A):
   def show(self):
      print("B")
class C(A):
   def show(self):
      print("C")
class D(B, C):
   pass
d = D()
d.show()          # Follows MRO: Output → B
print(D.__mro__)     # Shows method
resolution order
```

super() Function

Used to call the **parent class method** in a child class.

```
class Parent:
   def show(self):
      print("Parent class")
        class Child(Parent):
           def show(self):
              super().show()
              print("Child class")

        c = Child()
        c.show()
```

Polymorphism :- "many forms"

In OOP, it allows **methods/functions to behave differently** based on the object or context.
Same name, different **implementation** depending on the object/class.

Why Use Polymorphism?

Improves flexibility and scalability

Allows writing **general-purpose** code

Supports **runtime method resolution**

Types of Polymorphism in Python:

Type	**Example**
1. Duck Typing	Same method called on different types, Dynamic method resolution
2. Method Overriding	Child class overrides method of parent, Subclass changes parent behavior
3. Function Overloading (Not native)	Can be simulated using default args or @singledispatch
4. Operator Overloading	Redefine how operators behave on custom objects, Custom meaning to +, -, etc

Real Life Analogy

A **smartphone** uses the same "unlock" button:

- For **fingerprint**, **face ID**, or **password**.

The button is the same, but the behavior differs — that's polymorphism.

Method Overriding

Method Overriding is when a **child class redefines a method** that is already defined in its **parent class**. It is a key feature of **polymorphism** — allowing the **same method name** to behave **differently** depending on the object.

Why Use Method Overriding?

- To **change or extend** the behavior of a method from the parent class.
- To **customize behavior** in subclasses.

Real-Life Analogy

A **parent class** defines a general travel() method.
The **child class** overrides it to define specific travel modes (car, bike, flight).

Basic Example of Method Overriding

```
class Animal:
    def sound(self):
        print("Some animal sound")
class Dog(Animal):
    def sound(self):
        print("Bark!")
class Cat(Animal):
    def sound(self):
        print("Meow!")
a = Animal()
d = Dog()
c = Cat()
a.sound()
 # Output: Some generic animal sound
d.sound()  # Output: Bark!
c.sound()  # Output: Meow!
```

sound() is overridden in both Dog and Cat to provide specific behavior.

Using super() to Call Parent Method

You can call the parent class method from the child

class using super().

```
class Animal:
    def sound(self):
        print("Animal sound")
class Dog(Animal):
    def sound(self):
        super().sound()      # Call parent version
        print("Dog barks")
d = Dog()
d.sound()
# Output:
# Animal sound
# Dog barks
```

Example with Parameters

```
class Shape:
    def area(self, length, width):
        print("Area from Shape:", length * width)
class Square(Shape):
    def area(self, length, width):
        print("Area from Square:", length * width)
s = Square()
s.area(5, 5)  # Area from Square: 25
```

SOLID Principles:

S — Single Responsibility Principle (SRP)

A class should have only one reason to change.
Means: Each class/module/function should do **only one thing**.

```
 class Invoice:
   def __init__(self, amount):
      self.amount = amount
def calculate_total(self):
      return self.amount * 1.18  # GST 18%
# Violates SRP if we add printing or saving here
class InvoicePrinter:
   def print_invoice(self, invoice):
      print(f"Total: {invoice.calculate_total()}")
class InvoiceSaver:
   def save_to_db(self, invoice):
      print("Saved to database")
```

O — Open/Closed Principle (OCP)

Means: You should be able to add new behavior without changing existing code.

```
from abc import ABC, abstractmethod
class PaymentProcessor(ABC):
   @abstractmethod
   def process_payment(self, amount):
      pass
class CreditCardPayment(PaymentProcessor):
   def process_payment(self, amount):
      print(f"Paid {amount} via Credit Card")
class PayPalPayment(PaymentProcessor):
   def process_payment(self, amount):
      print(f"Paid {amount} via PayPal")
def pay(processor: PaymentProcessor, amount: float):
   processor.process_payment(amount)
```

L — Liskov Substitution Principle (LSP)

Subclasses should be substitutable for their base classes.
Means: Derived classes must not break the behavior expected from the base class.

```
class Bird:
   def fly(self):
      print("Flying")
class Sparrow(Bird):
   def fly(self):
      print("Sparrow flying")
class Ostrich(Bird):
   def fly(self): # Ostrich can't fly –breaks LSP
         raise NotImplementedError("Ostrich can't fly")
```

```
#  Fix:
class Bird:
   pass
class FlyingBird(Bird):
   def fly(self):
      pass
class Sparrow(FlyingBird):
   def fly(self):
      print("Sparrow flying")
class Ostrich(Bird):
   pass
```

I — Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use.
Means: Break big interfaces into smaller, role-specific ones.

```
class Workable:
   def work(self):
      pass
class Eatable:
   def eat(self):
      pass
class Human(Workable, Eatable):
   def work(self):
      print("Working")
def eat(self):
      print("Eating")
class Robot(Workable):
   def work(self):
      print("Robot working")
```

D — Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.
Means: Depend on interfaces, not concrete implementations.

```
class LightBulb:
   def turn_on(self):
      print("Light On")
def turn_off(self):
      print("Light Off")
# High-level module
class Switch:
   def __init__(self, device):
      self.device = device
def operate(self):
      self.device.turn_on()
# Using abstraction
bulb = LightBulb()
switch = Switch(bulb)
switch.operate()
```

ORM(Object Relational Mapping):-

It is a programming technique used to **map database tables to Python classes**, and **rows to Python objects**.

You interact with the database using **Python objects** instead of writing raw SQL queries.

- ORM connects Python objects with database tables.
- SQLAlchemy is the most powerful ORM in Python.
- You define tables as Python classes.
- You perform operations (insert/update/delete/query) using objects.
- Great for rapid development, especially in web apps and APIs.

<table>
<tr><th>Why Use ORM?</th><th>Popular Python ORMs</th><th>How ORM Works</th></tr>
<tr>
<td>
<table>
<tr><th>Feature</th><th>Benefit</th></tr>
<tr><td>Abstraction</td><td>No need to write raw SQL queries</td></tr>
<tr><td>Object-Oriented Approach</td><td>Work with classes and objects</td></tr>
<tr><td>Maintainability</td><td>Easier to manage and scale projects</td></tr>
<tr><td>Security</td><td>Prevents SQL injection attacks</td></tr>
<tr><td>Portability</td><td>Switch between databases easily</td></tr>
</table>
</td>
<td>• <b>SQLAlchemy</b> – Most powerful and widely used.
• <b>Django ORM</b> – Comes with Django framework.
• <b>Peewee</b> – Lightweight ORM.
• <b>Tortoise ORM</b> – Async support.</td>
<td>• A <b>Python class</b> represents a <b>table</b>.
• Each <b>class attribute</b> represents a <b>column</b>.
• Each <b>instance</b> of the class is a <b>row</b> in the table.</td>
</tr>
</table>

SQLAlchemy ORM Example

Let's create a simple database with users.

Install SQLAlchemy	Basic Setup	Create/Add a New Record

pip install sqlalchemy	from sqlalchemy import Column, Integer, String, create_engine
	from sqlalchemy.ext.declarative import declarative_base
	from sqlalchemy.orm import sessionmaker
	# Create a base class
	Base = declarative_base()
	# Define a class that maps to a table
	class User(Base):
	    __tablename__ = 'users'
	
	    id = Column(Integer, primary_key=True)
	    name = Column(String)
	    email = Column(String)
	# Create an SQLite database
	engine = create_engine('sqlite:///users.db', echo=True)
	Base.metadata.create_all(engine)
	# Create session
	Session = sessionmaker(bind=engine)
	session = Session()
		new_user = User(name='Ashish', email='ashish@example.com')
		session.add(new_user)
		session.commit()

Query Records	Update a Record	Delete a Record

users = session.query(User).all()
for user in users:
    print(user.id, user.name, user.email)
	user = session.query(User).filter_by(name='Ashish').first()
	user.email = 'ashish@newmail.com'
	session.commit()
		session.delete(user)
		session.commit()

ORM Structure Mapping

SQL Term	ORM Equivalent
Table	Class
Column	Attribute
Row	Object
Primary Key	primary_key=True
Foreign Key	ForeignKey()

ORM vs Raw SQL

Feature	ORM	Raw SQL
Abstraction	High	Low
Performance	Slightly slower	Faster
Flexibility	Easier to learn	More control
Learning Curve	Easier	Harder (requires SQL knowledge)

When to Use ORM

Use ORM if:

- You are building CRUD apps.
- You want rapid development.
- You like working with objects.

Avoid ORM if:

- You need raw performance (e.g. big data pipelines).
- You have complex queries with heavy optimization.

ORMs with MySQL/PostgreSQL

You can use SQLAlchemy with other databases by just changing the connection string:

For MySQL	engine = create_engine('mysql+pymysql://user:password@localhost/dbname')
For PostgreSQL	engine = create_engine('postgresql://user:password@localhost/dbname')

Programming Languages

Concept	Imperative Programming	Declarative Programming
Focus	*How* to do something (step-by-step)	*What* to do (final result)
Control	Gives full control to the developer	Lets the system handle details
Style	Command-driven (statements/loops)	Expression-driven (rules/logic)
Examples	C, Java, Python (in imperative style)	SQL, HTML, Prolog, Haskell
Use Case	System Programming, Game Development	Business rules/data, Web layout, Querying database

Imperative Programming

You write code that **tells the computer exactly what to do, step by step**.

Features:

- Uses variables, loops, conditionals.
- Closer to machine logic.
- Requires managing state (memory, flow, etc.).

Example In Python

```
# Sum of numbers from 1 to 5
total = 0
for i in range(1, 6):
   total += i
print(total)
```

Declarative Programming

You write **what result you want**, not how to get it.

Features:

- No explicit flow control.
- More abstract and readable.
- System (or engine) figures out the steps.

Example in SQL

```
SELECT SUM(number) FROM numbers_table;
```

Example in python

```
# Using built-in `sum` and list comprehension
print(sum([i for i in range(1, 6)]))
```

Analogy

Imagine making tea

- **Imperative:**
  "Boil water → Add tea leaves → Steep → Pour → Add sugar"
- **Declarative:**
  "Make me a cup of sweet tea"

Dunder Methods:-

Dunder = "Double UNDERSCORE" (e.g., __init__, __str__, __len__)

- They **start and end with double underscores**
- Python uses these methods to give special behavior to classes.
- They make your objects behave like **built-in types** (integers, lists, etc.)

Why Use Dunder Methods?

- Make custom objects behave like built-ins
- Cleaner, more Pythonic code
- Enable operator overloading (+, ==, etc.)
- Support iteration, context management, etc.

Examples of Common Dunder Methods

Method	**Purpose**	**Example Usage**
__init__	Object constructor (initializer)	obj = MyClass()
__str__	String representation (user-friendly)	print(obj)
__repr__	Official representation (debugging)	repr(obj)
__len__	Length of object	len(obj)
__getitem__	Get item using []	obj[key]
__setitem__	Set item using []	obj[key] = value
__delitem__	Delete item using del	del obj[key]
__iter__	Make object iterable	for i in obj:
__next__	Get next item in iteration	next(obj)
__eq__	Equality comparison (==)	obj1 == obj2
__lt__	Less than (<)	obj1 < obj2
__add__	Addition (+)	obj1 + obj2
__contains__	in operator	x in obj
__call__	Make object callable like a function	obj()
__del__	Destructor (object cleanup)	del obj

Custom Class Using Dunder Methods

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):
        return f"{self.title} ({self.pages} pages)"

    def __len__(self):
        return self.pages

    def __eq__(self, other):
        return self.pages == other.pages
```

Usage:

```
book1 = Book("Python Basics", 300)
book2 = Book("Advanced Python", 300)

print(book1)          # Python Basics (300 pages)
print(len(book1))     # 300
print(book1 == book2)  # True (because pages are equal)
```

Custom Operator Overloading

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"
```

Usage:

```
v1 = Vector(2, 3)
v2 = Vector(1, 4)
v3 = v1 + v2
print(v3)  # (3, 7)
```

What are __repr__ and __str__?

__repr__(self)	__str__(self)
• Official string representation of the object. • **Used for developers/debugging**. • Should be **unambiguous** and ideally return a string that could recreate the object. • Shows the name of the datatype and all arguments • Another roperty is that a programmer can normally use it to recreate an object equal to original one. • Aimed at the programmer	• Informal/pretty string representation. • **Used for users/output display**. • Should be **readable and friendly**. • Used for displaying information in a clean and understandable format • Aimed at user

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"

    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Usage:

```
p = Person("Ashish", 25)

print(p)          # Uses __str__:   Ashish is 25 years old
print(str(p))      # Uses __str__:   Ashish is 25 years old
print(repr(p))     # Uses __repr__:  Person(name='Ashish', age=25)
```

If Only __repr__ is Defined:

```
class Car:
    def __init__(self, brand):
        self.brand = brand

    def __repr__(self):
        return f"Car('{self.brand}')"

car = Car("Toyota")
print(car)         # Car('Toyota') → falls back to __repr__
```

GIL(Global Interpreter Lock**):-**

What is GIL?

The **Global Interpreter Lock (GIL)** is a **mutex (mutual exclusion lock)** that **allows only one thread to execute Python bytecode at a time**, even if you have multiple threads in your program.

Why does GIL exist?

CPython (the default and most widely used Python implementation) uses **reference counting** for memory management. Reference counting is **not thread-safe**, so the GIL was introduced to:

- **Protect memory** and prevent data corruption.
- Avoid the complexity of implementing fine-grained locking in the interpreter.

Example Problem with GIL

Even if you run Python code with **multiple threads**, only **one thread runs Python bytecode at a time**, while others are paused—even on multi-core CPUs.

```
import threading
def count():
    x = 0
    for i in range(10**7):
        x += 1
thread1 = threading.Thread(target=count)
thread2 = threading.Thread(target=count)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

Expected: Threads run in parallel on two cores, so faster.
Actual (in CPython): They run *one after another* due to the GIL.

When GIL Doesn't Matter

The GIL **only affects CPU-bound tasks**, not I/O-bound ones. In **I/O-bound tasks** (e.g., web scraping, file reading, network requests), threads often **release the GIL** while waiting for I/O

Good Use of Threads (I/O-bound):

```
import threading
import requests

def download(url):
    response = requests.get(url)
    print(f"{url} downloaded")

urls = ["https://example.com"] * 5
threads = [threading.Thread(target=download, args=(url,)) for url in urls]

for t in threads: t.start()
for t in threads: t.join()
```

Here, threads work well because while one thread waits for a network response, others can run.

How to Bypass the GIL?

- **Use Multiprocessing:**
  - ○ The multiprocessing module spawns **separate processes**, each with its **own Python interpreter and GIL**.
  - ○ Great for CPU-bound tasks.

```
from multiprocessing import Process
def compute():
    # heavy computation
    pass
p1 = Process(target=compute)
p2 = Process(target=compute)
p1.start()
p2.start()
p1.join()
p2.join()
```

- **Use C Extensions:**
  - ○ Some C libraries (like NumPy) do computations outside the GIL.
  - ○ This allows **parallel execution** in background threads.
- **Alternative Python Implementations:**
  - ○ **Jython**: No GIL (runs on Java VM).
  - ○ **IronPython**: No GIL (runs on .NET).
  - ○ **PyPy (STM version)**: GIL-free versions are experimental.

Feature	With GIL(Thread)	Without Gil(Multiprocessing)
CPU Utilization	Low (1 core)	High (multiple cores)
Overhead	Low	Higher (inter-process communication)
Suitable for	I/O-bound	CPU-bound
Sharing Data	Easy	Hard (requires serialization)

Final Notes

- GIL is a **CPython implementation detail**, not part of Python language itself.
- It simplifies interpreter design but limits concurrency in CPU-bound programs.
- Python is still **great for concurrent programming**, especially using **asyncio** or **multiprocessing** depending on the task.

Multitasking in Python:-

Multitasking refers to the ability of a program (or the operating system) to execute **multiple tasks (processes or threads) at the same time**. In Python, multitasking helps improve performance when dealing with I/O-bound or CPU-bound operations.

Types of Multitasking

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

Type	Description	Python Modules
Multithreading	Multiple threads within a single process	threading, concurrent.futures.ThreadPoolExecutor
Multiprocessing	Multiple processes (each with its own Python interpreter and memory space)	multiprocessing, concurrent.futures.ProcessPoolExecutor
Async I/O	Single-threaded but non-blocking using async / await. Uses **event loop** to handle many I/O tasks efficiently**.**	asyncio

Key Concepts:-

Thread	Process
• Lightweight sub-process. • Shares memory with the main thread. • Good for **I/O-bound** tasks (e.g., reading files, network calls).	• Independent memory space. • Heavyweight; better for **CPU-bound** tasks (e.g., calculations, image processing).

Multithreading

- Runs multiple threads **concurrently**.
- Threads share memory (risk of race conditions).
- Affected by **GIL** (Global Interpreter Lock) → not useful for CPU-bound tasks.

Good For: Waiting tasks (like downloading 100 web pages).

```
import threading
import time

def task(name):
    print(f"Start {name}")
    time.sleep(2)
    print(f"End {name}")

t1 = threading.Thread(target=task, args=("A",))
t2 = threading.Thread(target=task, args=("B",))
t1.start()
t2.start()
```

Multiprocessing

- Runs **separate processes**, each with its own memory and Python interpreter.
- Bypasses GIL → suitable for **CPU-heavy tasks**.

Good For: Parallel processing (e.g., 8-core CPU for 8 image tasks)

```
from multiprocessing import Process
import time

def task(name):
    print(f"Start {name}")
    time.sleep(2)
    print(f"End {name}")

p1 = Process(target=task, args=("A",))
p2 = Process(target=task, args=("B",))
p1.start()
p2.start()
```

Async I/O (asyncio)

- Uses **single-threaded** concurrency with an **event loop**.
- Doesn't block for I/O. Switches tasks while waiting.

Good For: High-scale servers, socket communication, async APIs.

```
import asyncio

async def task(name):
    print(f"Start {name}")
    await asyncio.sleep(2)
    print(f"End {name}")

async def main():
    await asyncio.gather(task("A"), task("B"))

asyncio.run(main())
```

Main Use Cases

Type	Best For
Multithreading	I/O-bound tasks: file I/O, web scraping, DB queries
Multiprocessing	CPU-bound tasks: math, ML, image processing
Async I/O	High-performance I/O like web servers, socket apps

Which Should You Use?

Problem Type	Use This
Downloading 1000 URLs	**Multithreading** or **Async I/O**
Resizing 1000 high-res images	**Multiprocessing**
Real-time chat app or web server	**Async I/O**
Machine learning / data crunching	**Multiprocessing**

Feature	Multithreading	Multiprocessing	Async I/O (asyncio)
Threads/Processes	Threads	Processes	Coroutines
GIL Affected	Yes	No	Yes
Best for	I/O-bound	CPU-bound	I/O-bound (high scale)
Memory Sharing	Shared	Separate	Shared (single-threaded)
Complexity	Medium	High (inter-process)	Medium (needs async)
Performance Gain	Some (I/O only)	High (multi-core)	High (async I/O tasks)

Concurrency	Parallelism
Doing **many things at once** (but not necessarily **simultaneously**). • **Definition**: Concurrency is when a system handles multiple tasks **by switching between them**, often rapidly. • **Example**: A single chef cooking multiple dishes by switching between them — cut vegetables, stir sauce, then check the oven. • **Used in**: Single-core systems using multitasking (threads, async I/O, etc.) • **Goal**: Improve responsiveness and resource usage. • **Python Example**: asyncio, threading	Doing **many things simultaneously** using **multiple processors or cores**. • **Definition**: Parallelism is when multiple tasks are **executed at the same time**. • **Example**: Multiple chefs each cooking one dish at the same time in different kitchen stations. • **Used in**: Multi-core CPUs, distributed systems. • **Goal**: Improve speed/performance. • **Python Example**: multiprocessing, joblib, concurrent.futures.ProcessPoolExecutor

Analogy

Feature	Concurrency	Parallelism
What it means	Dealing with many tasks at once	Doing many tasks at the same time
Hardware required	Can be single-core	Needs multi-core or multiple processors
Primary focus	Structure	Execution
Python modules	asyncio, threading	multiprocessing, joblib, ray

Functional Requirements

(What the system should do)

These are features and behaviors that define **specific functionality** of the system.

Examples:	Covers:
User can log in and log out, Admin can add or delete users, The system sends a confirmation email after registration, A customer can add items to a shopping cart, The app calculates total price with taxes	Business rules, User authentication, Data processing, APIs and system interactions

Non-Functional Requirements

(How the system performs tasks (qualities))

These define the **quality attributes**, performance, and constraints of the system — not specific behaviors, but **how well** it works.

Examples:	Covers:
The system must load within 2 seconds, Should support up to 10,000 users simultaneously, Must be available 99.9% of the time, Should be secure from SQL injection, Should run on Windows and Linux	Performance, Reliability, Scalability, Usability, Security, Compatibility

Context Switching

Context Switching is the process of **storing the state** of a currently running task (like a thread or process) and **restoring the state** of another task — so the CPU can switch between them efficiently.

Real-Life Analogy	In Computing Terms	Steps in Context Switching:
Imagine you're writing two different essays. You write a few lines in one, **pause**, save your place, and then start writing the second one. When you come back to the first, you **resume where you left off**.	• **Context**: Information like registers, program counter, memory state, etc. for a process. • **Switching**: CPU stops running Process A, saves its context, loads the context of Process B, and starts executing Process B.	• Save state of the current process (registers, stack pointer, etc.) • Update PCB (Process Control Block) of the current process • Load state of the new process from its PCB • Start/resume execution of the new process

Where It Happens	Context Switch Overhead	Context Switching Time
Between two **threads** Between two **processes** In **multitasking**, **multithreading**, and **concurrent systems**	Context switching isn't free — it **uses CPU time**. Too much switching (called **thrashing**) can reduce performance.	Measured by the time it takes to: Save the current context Load the new context Good OS designs try to **minimize** this time for better efficiency.

Example in Python (Threading):

```
import threading
def task():
    for _ in range(5):
        print(f"Task running in
{threading.current_thread().name}")
# Create two threads
t1 = threading.Thread(target=task, name="Thread-1")
t2 = threading.Thread(target=task, name="Thread-2")
t1.start()
t2.start()
t1.join()
t2.join()
```

The CPU **switches between Thread-1 and Thread-2**, using context switching.

enumerate() function:-

The enumerate() function in Python is used when you want to **loop through a list (or any iterable) and get both the index and the value** at the same time.

Syntax	**enumerate(iterable, start=0)**

- **iterable**: the sequence you want to iterate over (like a list, tuple, string, etc.)
- **start**: optional, default is 0. It sets the starting index.

Example 1: Basic Usage

```
fruits = ['apple', 'banana', 'cherry']
for index, value in enumerate(fruits):
    print(index, value)
```

Output:

```
0 apple
1 banana
2 cherry
```

Example 2: Start index from 1

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

Output:

```
1 apple
2 banana
3 cherry
```

Without enumerate (less elegant way)

```
for i in range(len(fruits)):
    print(i, fruits[i])
```

Why use enumerate()?

- It makes code **cleaner** and more **Pythonic**
- Avoids manually tracking indexes

What is a Generator..?

- In Python, a **generator** is a special type of **iterator** that **yields values one at a time** using the yield keyword **instead of returning them all at once** like a regular function.
- Generators are **memory-efficient**, especially useful when working with **large data streams or infinite sequences**, because they **generate values on the fly** (lazy evaluation).
- **A generator is:**
  - A **function** that contains the yield statement.
  - Returns an **iterator**, which can be iterated using next() or a loop.
  - Does **not store the entire sequence in memory**.

Creating a Generator Function

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1
```

Using the generator

```
gen = count_up_to(3)
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
print(next(gen))  # StopIteration error
```

You can also use a for loop

```
for num in count_up_to(3):
    print(num)
```

Comparison: Generator vs List

List function:

```
def get_numbers_list(n):
    return [i for i in range(n)]
```

Stores all numbers in memory.

Generator version:

```
def get_numbers_gen(n):
    for i in range(n):
        yield i
```

Does not store all numbers — generates one at a time.

Generator Expressions

Just like list comprehensions but with () instead of [].

```
gen = (x * x for x in range(5))
for i in gen:
    print(i)
```

How it works

- The first time you call the function, it **returns a generator object**.
- Each time you call next(), the generator **resumes** where it left off and **executes until it hits yield** again.
- Once all yields are done, it raises StopIteration.

Advantages

- Memory efficient (no need to store all values)
- Represent infinite streams (like Fibonacci, file reading line by line)
- Cleaner code when working with iterators

Real-World Example: Reading Large Files

```
def read_file_line_by_line(filename):
    with open(filename) as f:
        for line in f:
            yield line.strip()
```

Example: Infinite Generator

```
def infinite_counter(start=0):
    while True:
        yield start
        start += 1
```

What is CV (Computer Vision) in Python?

- **Computer Vision (CV)** is a field of artificial intelligence that allows computers to **see, interpret, and process visual data (images/videos)** the same way humans do.
- In Python, **Computer Vision** is commonly implemented using the **OpenCV library** (cv2 module), which provides tools for:
  - Image & video processing
  - Object detection
  - Face recognition
  - Edge detection
  - Camera handling

Installing OpenCV

```
pip install opencv-python
```

Example: Read and Show an Image

```
import cv2
# Read an image
img = cv2.imread('sample.jpg')
# Show the image in a window
cv2.imshow('My Image', img)
# Wait until a key is pressed
cv2.waitKey(0)
# Close all OpenCV windows
cv2.destroyAllWindows()
```

Real-World Applications

Use Case	Description
Face Recognition	Attendance, unlocking devices
Object Detection	Self-driving cars, retail
OCR (Text Reading)	**S**canning documents, number plates
Augmented Reality	AR filters, gaming
Medical Imaging	Tumor detection, X-ray analysis

Common OpenCV Tasks

Reading and Writing Images

```
img = cv2.imread('input.jpg')
cv2.imwrite('output.jpg', img)
```

Resizing an Image

```
resized = cv2.resize(img, (300, 300))
```

Convert to Grayscale

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Edge Detection (Canny)

```
edges = cv2.Canny(gray, 100, 200)
```

Drawing on Image

```
cv2.rectangle(img, (50, 50), (200, 200), (255, 0, 0), 2)
cv2.putText(img, 'Label', (50, 40),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
```

Face Detection Using Haar Cascade

```
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5)
for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)
```

Video Capture Using Webcam

```
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    cv2.imshow('Webcam', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

What Are Libraries in Python?

In Python, a **library** is a collection of **pre-written code** (modules and packages) that provides **functions, classes, and tools** for common tasks so you don't have to write everything from scratch.

Why Use Libraries?

- Save time and effort
- Improve code readability
- Solve complex problems with fewer lines
- Reuse tested and optimized code

Examples of Common Python Libraries

Area	Library	Purpose
Math & Numbers	math, random, decimal	Basic to advanced mathematics
Data Analysis	pandas, numpy	Dataframes, arrays, calculations
Visualization	matplotlib, seaborn, plotly	Charts and graphs
Machine Learning	scikit-learn, xgboost	ML algorithms
Deep Learning	tensorflow, keras, torch	Neural networks
Web Development	flask, django, fastapi	Web servers and APIs
Automation	os, shutil, subprocess	File and system automation
Web Scraping	requests, beautifulsoup4, selenium	Fetching data from web
Computer Vision	opencv-python (cv2)	Image & video processing
Natural Language Processing	nltk, spacy, transformers	Text analysis and AI

Importing Libraries

Built-in Library
Comes with Python (e.g., math, datetime, os)

```
import math
print(math.sqrt(16))
```

External Library (e.g. NumPy)
Installed via pip (e.g., pandas, flask, opencv-python)
You must install it first:

```
pip install numpy
```

Then use it:

```
import numpy as np
arr = np.array([1, 2, 3])
print(np.mean(arr))
```

Python Libraries for Data Science

Category	Library	Purpose
Data Manipulation	pandas	Tabular data analysis (DataFrame), filtering, grouping, merging
	numpy	Numerical computing, arrays, matrices, statistics
Visualization	matplotlib	Basic plotting: line, bar, scatter, histogram
	Seaborn	Statistical plots: heatmaps, boxplots, violin plots
	plotly, bokeh	Interactive, web-based visualizations
Machine Learning	scikit-learn (sklearn)	ML algorithms: classification, regression, clustering, evaluation
	xgboost, lightgbm	Fast gradient boosting for tabular data
Deep Learning	tensorflow, keras	Neural networks, deep learning, image & text processing
	pytorch	Flexible deep learning for research and production
NLP (Text Analysis)	nltk	Tokenization, stemming, stopwords
	spaCy	Industrial-level NLP: POS tagging, NER, dependency parsing
	transformers (HuggingFace)	Pretrained LLMs (BERT, GPT, etc.)
Web Scraping & APIs	requests	Make HTTP requests to web/API servers
	beautifulsoup4	Parse HTML/XML for data extraction
	selenium	Browser automation for scraping dynamic content
Data Cleaning & Prep	sklearn.preprocessing	Encoding, normalization, scaling, imputing
	missingno	Visualize and handle missing data
Statistics	scipy	Statistical functions, optimization, distributions
	statsmodels	Hypothesis testing, linear models, time series
Auto EDA	pandas-profiling, sweetviz	Generate automatic exploratory data analysis reports
	dtale, ydata-profiling	Live, interactive views of DataFrames

Pandas:-

pandas is one of the most powerful and widely-used **data analysis** and **data manipulation** libraries in Python. It is designed for working with **structured data** like tables, spreadsheets, or time-series data.

Why Use pandas?

- Easy to load, clean, transform, and analyze data
- Offers **DataFrame**, a flexible 2D data structure (like Excel)
- Built on top of NumPy, integrates well with other data science tools
- Efficient and fast even with large datasets

Key Data Structures in pandas

Structure	Description	Example Use
Series	1D labeled array (like a single column)	Storing a list of values
DataFrame	2D labeled table (rows & columns)	Spreadsheets, CSV files

Basic Example

```
import pandas as pd
# Create a DataFrame
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['Delhi', 'Mumbai', 'Chennai']
}
df = pd.DataFrame(data)
print(df)
```

Output:

```
     Name    Age    City
0    Alice    25    Delhi
1    Bob      30    Mumbai
2    Charlie  35    Chennai
```

Common pandas Operations

Read Data

```
df = pd.read_csv("data.csv")        # From CSV
df = pd.read_excel("data.xlsx")     # From Excel
```

Explore Data

```
df.head()         # First 5 rows
df.info()         # Summary
df.describe()    # Stats for numerical columns
df.columns       # List column names
```

Select Data

```
df['Age']                  # Select column
df[['Name', 'City']]       # Select multiple columns
df.iloc[0]                 # Select first row by index
df.loc[0, 'Name']          # Value at row 0, column 'Name'
```

Add/Remove Columns

```
df['Salary'] = [50000, 60000, 70000]   # Add column
df.drop('Age', axis=1, inplace=True)   # Drop column
```

Filter Rows

```
df[df['Age'] > 28]                   # Filter condition
df[df['City'].str.contains('Del')]    # String match
```

Sort & Group

```
df.sort_values('Age', ascending=False)
df.groupby('City').mean()
```

Handle Missing Data

```
df.isnull().sum()        # Check missing values
df.fillna(0)            # Replace missing with 0
df.dropna()              # Drop rows with missing values
```

Export Data

```
df.to_csv("output.csv", index=False)      # Save to CSV
df.to_excel("output.xlsx", index=False)   # Save to Excel
```

Real-World Use Cases

Use Case	Example
Data Cleaning	Remove missing/duplicate entries
EDA (Analysis)	Describe trends, patterns, stats
Data Transformation	Convert, merge, encode, filter data
Reporting	Summarize and export analysis results

NumPy:-

NumPy (Numerical Python) is a **fundamental library** for **numerical computing** in Python. It provides:

- High-performance **multidimensional arrays**
- Tools for **mathematics**, **linear algebra**, **statistics**, **FFT**, etc.
- Basis for libraries like **pandas**, **scikit-learn**, **tensorflow**, and more

Why Use NumPy?

- Efficient array storage and computation
- Faster than regular Python lists
- Powerful vectorized operations (no loops!)
- Backbone for scientific and data computing in Python

Core Data Structure: ndarray

An ndarray (N-dimensional array) is like a **list of numbers**, but more powerful.

```
import numpy as np
arr = np.array([1, 2, 3])
print(arr)        # [1 2 3]
print(type(arr))  # <class 'numpy.ndarray'>
```

NumPy Array Types

Type	Example
1D array	np.array([1, 2, 3])
2D array	np.array([[1, 2], [3, 4]])
3D+ arrays	np.array([[[...]]])

NumPy Basics

Array Properties

```
a = np.array([[1, 2], [3, 4]])
a.shape     # (2, 2)
a.size      # 4
a.ndim      # 2 (2D)
a.dtype     # dtype('int64') or similar
```

Creating Arrays

```
np.zeros((2, 3))       # 2x3 array of 0s
np.ones((3, 3))        # 3x3 array of 1s
np.full((2, 2), 7)     # 2x2 array filled with 7
np.eye(3)              # Identity matrix
np.arange(0, 10, 2)    # [0 2 4 6 8]
np.linspace(0, 1, 5)   # [0.  0.25 0.5 0.75 1. ]
```

Array Operations

Element-wise Operations

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
a + b     # [5 7 9]
a * b     # [4 10 18]
a ** 2    # [1 4 9]
```

Matrix Operations

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])
np.dot(A, B)      # Matrix multiplication
A.T               # Transpose
np.linalg.inv(A)  # Inverse (if square & non-singular)
```

Indexing, Slicing, Filtering

```
a = np.array([10, 20, 30, 40, 50])
a[0]       # 10
a[1:4]     # [20 30 40]
a[a > 30]  # [40 50]
```

Useful NumPy Functions

Function	Purpose
np.mean(arr)	Mean of array
np.median(arr)	Median
np.std(arr)	Standard deviation
np.sum(arr)	Sum of elements
np.max(arr), np.min(arr)	Max/Min
np.sort(arr)	Sort array
np.unique(arr)	Get unique values

Advantages Over Lists

Python List	NumPy Array
Slower, needs loops	Fast vectorized operations
Can store mixed types	Stores homogeneous data
No advanced math functions	Supports broadcasting, linear algebra

Matplotlib:-

- **matplotlib** is a popular **data visualization library** in Python.
- It allows you to create a wide variety of **static**, **animated**, and **interactive plots** — just like charts in Excel or Google Sheets.

Common Plot Types in Matplotlib

Plot Type	Function	Use Case
Line Plot	plt.plot()	Trend over time or sequence
Bar Chart	plt.bar()	Category comparison
Horizontal Bar	plt.barh()	Category comparison (horizontal)
Pie Chart	plt.pie()	Part-to-whole (percentages)
Scatter Plot	plt.scatter()	Relationship between two variables
Histogram	plt.hist()	Frequency distribution
Box Plot	plt.boxplot()	Outliers and quartiles

Example:-

Line Plot

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]
plt.plot(x, y)
plt.title("Simple Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Bar Chart

```
categories = ['A', 'B', 'C']
values = [10, 25, 15]

plt.bar(categories, values)
plt.title("Bar Chart Example")
plt.show()
```

Customize Your Plots:-

```
plt.plot(x, y, color='red', marker='o', linestyle='--')
plt.title("Styled Plot")
plt.xlabel("Time")
plt.ylabel("Value")
```

You can customize:

- color='green'
- linestyle='--'
- marker='o', 's', '*', etc.
- linewidth=2

Multiple Plots on Same Chart

```
plt.plot(x, y, label='Line 1')
plt.plot(x, [i*2 for i in y], label='Line 2')

plt.legend()       # Show legend
plt.title("Multiple Lines")
plt.show()
```

Subplots (Multiple Charts in One Figure)

```
fig, axs = plt.subplots(1, 2)  # 1 row, 2 columns

axs[0].plot(x, y)
axs[0].set_title('Plot 1')

axs[1].bar(x, y)
axs[1].set_title('Plot 2')

plt.tight_layout()
plt.show()
```

Save Plot to File	plt.savefig("my_plot.png", dpi=300)

Integration: Matplotlib works great with:

- **NumPy** arrays
- **Pandas** DataFrames (df.plot())
- **Seaborn** (built on top of matplotlib)

Seaborn:-

Seaborn is a **statistical data visualization** library built on top of **Matplotlib**.
It provides **beautiful, easy-to-use, and informative plots** with **less code** and better **default aesthetics**.

Why Use Seaborn?

- High-level API for statistical plots
- Works directly with **pandas DataFrames**
- Attractive **default styles and color palettes**
- Built-in functions for **regression**, **categorical**, and **matrix plots**

Installation	pip **install** seaborn

Then import it:	**import** seaborn **as** sns
	import matplotlib.pyplot **as** plt

Example Dataset (Seaborn Built-in)

```
import seaborn as sns
df = sns.load_dataset("tips")  # Load dataset
df.head()
```

Common Seaborn Plot Types:

Plot Type	Function	Use Case
Line Plot	sns.lineplot()	Trend analysis
Bar Plot	sns.barplot()	Categorical mean comparisons
Count Plot	sns.countplot()	Frequency counts of categories
Box Plot	sns.boxplot()	Distribution with outliers
Violin Plot	sns.violinplot()	Box + distribution view
Strip/Swarm Plot	sns.stripplot() / swarmplot()	Scatter for categories
Scatter Plot	sns.scatterplot()	Correlation between two variables
Regression Plot	sns.regplot()	Scatter + regression line
Pair Plot	sns.pairplot()	Matrix of scatterplots
Heatmap	sns.heatmap()	Correlation matrix or pivot tables

Examples:

Bar Plot

```
sns.barplot(x="day", y="total_bill", data=df)
plt.title("Average Bill by Day")
plt.show()
```

Box Plot

```
sns.boxplot(x="day", y="total_bill", data=df)
```

Pair Plot

```
sns.pairplot(df, hue="sex")
```

Count Plot

```
sns.countplot(x="sex", data=df)
```

Heatmap

```
corr = df.corr(numeric_only=True)
sns.heatmap(corr, annot=True, cmap="coolwarm")
```

Customization in Seaborn

```
sns.set_style("whitegrid")        # Background grid
sns.set_palette("pastel")         # Color palette
sns.boxplot(x="day", y="tip", data=df)
plt.title("Tips by Day")
```

Seaborn vs Matplotlib

Feature	Seaborn	Matplotlib
Syntax	High-level	Low-level
Aesthetics	Attractive by default	Requires manual styling
Data Input	Works well with pandas	Needs manual data formatting
Statistical Plots	Built-in	Must be created manually

Misssing Values detection using pandas & numpy

To **detect missing values** using **Pandas** and **NumPy**, you typically work with functions like isnull(), notnull(), isna(), isnan(), and others. Below is a detailed explanation with examples:

Using Pandas

isnull() or isna():-

Both do the same: detect missing values (NaN, None) and return a DataFrame of booleans.

```
import pandas as pd
# Sample DataFrame
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', None],
    'Age': [25, None, 22],
    'City': ['New York', 'London', 'Delhi']
})
print(df.isnull())
```

Get rows with missing values

```
print(df[df.isnull().any(axis=1)])
```

Count missing values per column

```
print(df.isnull().sum())
```

Check if any missing values in DataFrame

```
print(df.isnull().values.any())  # True if any missing
```

Using NumPy

NumPy can be used if you're working with arrays or combining with pandas.

```
import numpy as np
# Example array
arr = np.array([1, 2, np.nan, 4])
# Detect NaN
print(np.isnan(arr))  # [False False  True False]
```

You can also combine with pandas:

# Detect using numpy in a pandas DataFrame

```
print(np.isnan(df['Age']))
```

(Note: This works only for numeric columns; for object types use pd.isnull().)

Comparing Pandas and NumPy Detection Functions

Task	Pandas	NumPy
Check missing values	pd.isnull()	np.isnan()
Count missing values	df.isnull().sum()	np.isnan(array).sum()
Check if any value is missing	df.isnull().any()	np.isnan(array).any()
Filter rows with missing values	df[df.isnull().any(1)]	Not directly possible (requires DataFrame)
To find **non-missing** values:	pd.notnull(df)	~np.isnan(array)

Outlier Detection:-

Outlier detection is the process of identifying data points that significantly deviate from the rest of the dataset. You can detect outliers using **Pandas**, **NumPy**, and optionally **visualizations** with libraries like **Matplotlib**, **Seaborn**, or **Plotly**.

Common Outlier Detection Techniques:

Using IQR (Interquartile Range) Method:-

Outliers are typically:

- Below **Q1 - 1.5×IQR**
- Above **Q3 + 1.5×IQR**

```
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({
   'Score': [12, 14, 15, 16, 18, 21, 22, 90, 95]
})
# IQR method
Q1 = df['Score'].quantile(0.25)
Q3 = df['Score'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

outliers = df[(df['Score'] < lower_bound) | (df['Score'] > upper_bound)]
print(outliers)
```

Using Z-Score (Standard Deviation):-

Outliers have a **Z-score > 3** or **< -3** (commonly used threshold).

```
from scipy.stats import zscore
import numpy as np

df['z_score'] = zscore(df['Score'])

outliers_z = df[np.abs(df['z_score']) > 3]
print(outliers_z)
```

Using Boxplot (Visual Detection):-

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.boxplot(x=df['Score'])
plt.show()
```

Outliers appear as individual points outside the box whiskers.

Using NumPy Percentiles:-

Quick detection without Pandas:

```
import numpy as np

data = np.array([12, 14, 15, 16, 18, 21, 22, 90, 95])

q1 = np.percentile(data, 25)
q3 = np.percentile(data, 75)
iqr = q3 - q1

lower = q1 - 1.5 * iqr
upper = q3 + 1.5 * iqr

outliers_np = data[(data < lower) | (data > upper)]
print(outliers_np)
```

Method	Library	Criteria for Outlier
IQR	Pandas	Outside [Q1 - 1.5×IQR, Q3 + 1.5×IQR]
Z-Score	Scipy / Numpy	Z > 3 or Z < -3
Boxplot	Seaborn	Dots outside whiskers
Percentile	Numpy	Custom thresholds

Introduction to GenAI (Generative AI):-

Generative AI (GenAI) refers to a class of artificial intelligence models that are capable of generating **new content** — such as text, images, code, audio, and even video — based on the data they were trained on.

What Does Generative AI Do?

GenAI models **create** rather than simply **analyze**. They can:

- Write essays, stories, emails, and documentation
- Generate images or art from text prompts
- Produce realistic voice and music
- Write or debug computer code
- Generate summaries, quizzes, and answers from documents

How Does GenAI Work?

It's powered by **deep learning** techniques — especially:

- **Transformers** (e.g., GPT, BERT, T5)
- **Large Language Models (LLMs)** like ChatGPT, Gemini, Claude, LLaMA
- **Diffusion models** (for images, e.g., DALL·E, Stable Diffusion)

These models are trained on massive datasets and learn **patterns**, **structures**, and **relationships** in language, code, or pixels.

Applications of GenAI

Domain	Use Case Example
Content Writing	Blog posts, ads, social media captions
Programming	Code generation, explanation, bug fixes
Design & Art	AI-generated logos, illustrations, avatars
Education	AI tutors, quiz makers, resume feedback
Gaming	Character dialogue, story plots, textures
Data Science	Report generation, insights from data
Conversational AI	Chatbots, virtual assistants

Popular GenAI Tools & APIs

Tool/API	Purpose
OpenAI GPT	Text generation, coding
Google Gemini	Multimodal (text + image) generation
Claude (Anthropic)	Safer LLM assistant
DALL·E	Image generation from text
Stable Diffusion	Open-source image creation
LangChain	Build GenAI-powered apps
Hugging Face	Open-source model platform

Advantages

- Speeds up creative processes
- Reduces human workload
- Enhances personalization
- Works across multiple domains

Challenges

- Can generate **incorrect or biased** output
- May require **fine-tuning** for specific use
- Legal/ethical concerns with data usage

Example Use Case in Python (Text Generation)

```python
from openai import OpenAI
response = openai.Completion.create(
  engine="gpt-4",
  prompt="Write a short story about a robot who learns to paint",
  max_tokens=150
)
print(response.choices[0].text)
```

Comparison Table of Popular GenAI Platforms:-

Feature / Model	OpenAI GPT-4 / GPT-4o	Google Gemini (1.5 Pro)	Claude 3 (Anthropic)	LLaMA 3 (Meta)	Mistral / Mixtral
Model Type	Large Language Model (LLM)	Multimodal (Text, Image, Code)	Natural Language Assistant	Open-source LLM	Open-source LLM
Modalities	Text, Image, Code, Audio (GPT-4o)	Text, Image, Code, Video (v1.5)	Text, Code, Reasoning	Text, Code	Text, Code
Context Window	Up to 128K tokens (GPT-4o)	1M tokens (Gemini 1.5 Pro)	Up to 200K tokens	8K – 32K (depending on version)	32K – 65K
Open-Source	N (proprietary)	N (proprietary)	N (proprietary)	Y (open-source)	Y
Best Use Case	Chatbots, code, logic, writing	Multimodal, summarization, logic	Safe assistants, summarization	Research, open projects	Lightweight open-source models
APIs/SDKs Available	OpenAI API	Gemini API (Google AI Studio)	Claude API	via Hugging Face	via Hugging Face
Training Data Cutoff	Apr 2023 (GPT-4o: Oct 2023)	Mid-2024 (Gemini 1.5 Pro)	Aug 2023	Mar–Apr 2023	2023 (varies by model)
Website/Access	chat.openai.com	gemini.google.com	claude.ai	huggingface.co	huggingface.co

Introduction to LangChain:-

LangChain is a powerful **framework** designed to help developers build applications powered by **Large Language Models (LLMs)** like **GPT**, **Claude**, **Gemini**, etc., in a **modular, scalable, and production-ready** way.

It acts as the **"middleware"** to connect LLMs with **data sources**, **tools**, **memory**, **APIs**, **agents**, and more — enabling creation of **real-world GenAI applications** such as chatbots, document Q&A tools, RAG systems, assistants, and more.

Why LangChain?

Building with raw LLM APIs (e.g., OpenAI or Gemini) is limiting — you need to manage:

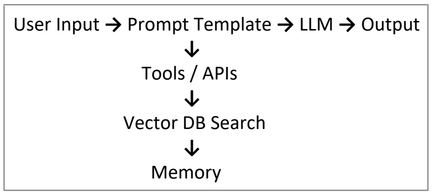
- Prompting logic
- Tool usage (e.g., search, calculator)
- Document parsing
- Memory / context
- Agent decision-making

LangChain helps with all of this in a clean, plug-and-play style.

LangChain Features at a Glance

Feature	Description
Prompt Templates	Create reusable, dynamic prompts
Chains	Combine multiple LLM calls / logic in a flow
Agents	Let the LLM choose which tool to use next (based on reasoning)
Retrieval (RAG)	Search documents using vector DBs like FAISS, Pinecone, Chroma
Memory	Remember past conversations (e.g., for chatbots)
Tool Integration	Add tools like calculators, search APIs, SQL, Python REPL, etc.
Multi-Model	Supports GPT, Gemini, Claude, Mistral, Cohere, HuggingFace, Ollama, etc.

LangChain App Architecture



Install LangChain

```
pip install langchain
```

And install a model wrapper like OpenAI:

```
pip install openai
```

LangChain Ecosystem

Tool / Integration	Use
OpenAI, Gemini, Claude	Supported as backends
FAISS, Chroma, Pinecone	Vector stores for Retrieval
Streamlit / Gradio	Frontends for demo apps
FastAPI / Flask	API-based LLM apps

Use Case Examples

App Name	Description
PDF Q&A Bot	Ask questions from a PDF or DOCX
Custom Chatbot	Memory + Tool use + Search
AI Resume Screener	Parse resumes and match jobs
SQL Generator	Convert user question → SQL → Execute on DB
RAG System	LLM + Vector DB search for document QA

Summary

Feature	LangChain Helps You With
Prompting	Reusable, safe, parameterized prompts
Chains	LLM workflows (multi-step)
Agents	Dynamic decision-making
Retrieval	Combine vector search + LLM (RAG)
Memory	Chat history, context retention
Tools	Use external functions / APIs

Example: Simple LLM Chain (OpenAI)

```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

llm = OpenAI(model_name="gpt-3.5-turbo")

prompt = PromptTemplate.from_template("Translate to French: {text}")
chain = LLMChain(llm=llm, prompt=prompt)

print(chain.run("Hello, how are you?"))
```

Example: Summarizer with GPT

```
from langchain.llms import OpenAI
from langchain.chains.summarize import load_summarize_chain
from langchain.document_loaders import TextLoader
llm = OpenAI(temperature=0)
loader = TextLoader("sample.txt")
docs = loader.load()

chain = load_summarize_chain(llm, chain_type="map_reduce")
summary = chain.run(docs)
print(summary)
```