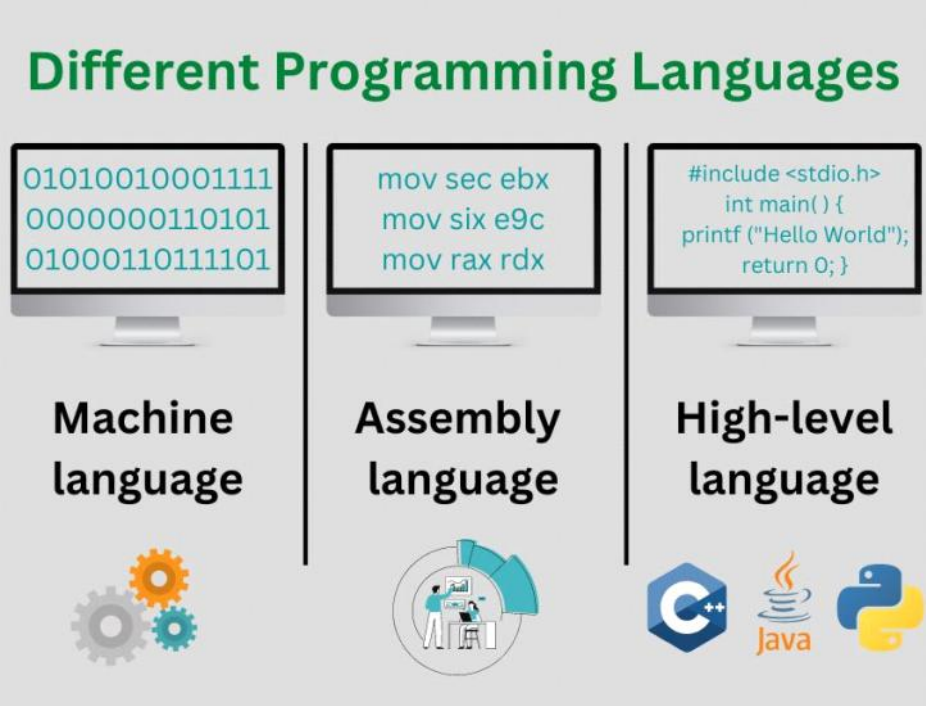
Programming Languages

A **programming language** is a formal language used to communicate instructions to a computer. These instructions tell the computer what to do, such as performing calculations, processing data, or interacting with hardware.



Python Code (High-level)
↓
Python Bytecode (Low-level) (platform independent)
↓
Machine Code (Binary 0s and 1s)
↓
CPU Executes

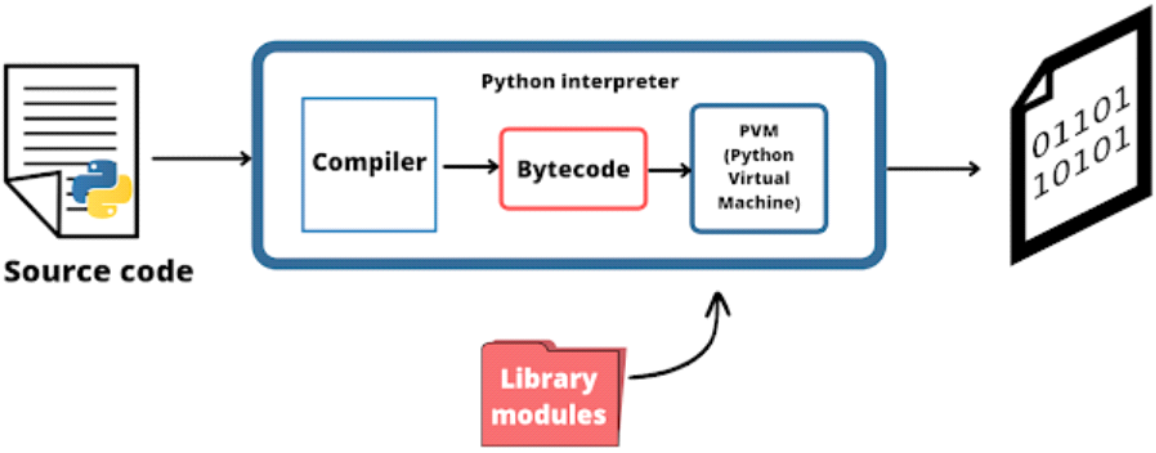
🔁 What Happens When You Run a Python Program?

1. **You write Python code** (high-level language):

   print("Hello, world!")

2. **Python interpreter converts it to bytecode (an intermediate, low-level form)**:
   - Bytecode is not machine code yet, but it's closer than Python.
   - It's platform-independent and runs on the **Python Virtual Machine (PVM)**.
   - Example (conceptually): LOAD_NAME 'print', LOAD_CONST 'Hello, world!', CALL_FUNCTION

3. **The Python Virtual Machine (PVM)** executes the bytecode:
   - Under the hood, the PVM uses a mix of C code and system-level instructions.
   - This eventually gets translated into machine code that the CPU understands (via the interpreter, or sometimes via a Just-In-Time (JIT) compiler in implementations like PyPy).


Python interpreter
Compiler
Bytecode
PVM
(Python
Virtual
Machine)
Source code
01101
10101
Library
modules

Python, Variables ,Data types ,operators and if else

What is Python

Install Python from https://www.python.org/downloads/

(Make sure to check “Add Python to PATH” during installation ☑)

https://code.visualstudio.com/download ,Install python extension

Python is a **high-level, Interpreted , general-purpose programming language** known for its **simplicity** and **dynamic typing**.

1- **High level :** A **high-level language** is a **programming language designed to be easy for humans to read, write, and understand**, abstracting away the complex details of the computer's hardware. Python converts it to bytecode - a low level platform independent representation.(Mac, Windows , Linux etc)

Example: Python (High-Level) vs Assembly (Low-Level)

High-Level (Python):

```
python
x = 10
y = 20
z = x + y
print(z)
```

Low-Level (Assembly - pseudocode):

```
 MOV AX, 1
 MOV BX, 20
 ADD AX, BX
 CALL PRINT_AX
```

2- **Interpreted :** An **interpreted language** is a programming language where code is **executed line-by-line by an interpreter at runtime**, **without being compiled into machine code ahead of time**.

```
 print("Hello!")
 x = 5 + 3
 print(x)
```

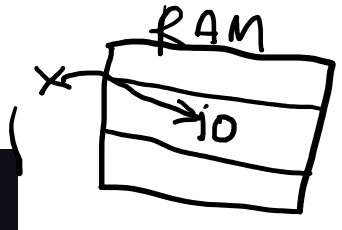
1. Reads print("Hello!") → prints Hello!
2. Reads x = 5 + 3 → calculates 8 and assigns it
3. Reads print(x) → prints 8

Each line is read, executed, and then discarded, rather than compiled all at once.

Variables

A **variable in Python** is a **name** that refers to a **value stored in memory**.

It’s like a label you stick on a box to remember what's inside.




```
x = 10
name = "Ankit"
is_active = True
```

Here:

- x is a variable holding the integer 10
- name holds the string "Ankit"
- is_active holds the boolean True

Variables are stored in RAM and their location can be printed using id(variable name)

```
print(id(name))
```

Variables with the same value will have same address on RAM.

Key Points about Python Variables:

Feature	Description
Case-sensitive	name and Name are different variables
Must start with	A letter or underscore (_), not a number
Can hold any type	Strings, numbers, lists, functions, objects, etc.
Dynamically typed	A variable's type can change at runtime
No need to declare	You don't have to specify the type (e.g., int, string)

Data Types in Python

A data type in Python (or any programming language) tells the computer what kind of value a variable holds and what operations can be performed on it.

Data Type	Type Name	Example	Description
Integer	int	10, -5	Whole numbers
Float	float	3.14, 0.0	Decimal (floating-point) numbers
String	str	"Hello"	Sequence of characters (text)
Boolean	bool	True, False	Logical values (yes/no)

```
x = 10        # int (integer)
name = "Ankit" # str (string)
pi = 3.14     # float (decimal)
is_on = True  # bool (boolean)
```

You can check the type of variable using function type(variable name).

3- Dynamically typed :

A **dynamic type** means that the **type of a variable is determined at runtime**, not when you write the code.

In Python, you don't have to declare the data type of a variable — Python figures it out automatically when the code runs.

Dynamic Typing vs Static Typing

Feature	Dynamic Typing (Python)	Static Typing (C, Java)
Type declaration	Not needed	Required
Type checked at	Runtime	Compile time
Flexible	☑ Yes	✗ No (strict types)
Example	x = 10	int x = 10;

Print Statement

The print() function in Python is used to display output to the screen.

Use Cases:

Purpose	Example
Show variable value	print(name)
Display message	print("Welcome!")
Debug your code	print("Step 1 done")
Print results	print("Total:", total_area)

Add new line using \n or enter multiline text in 3 single quotes

```
# string method
print("The length of my rectangle is",length,"and breadth is" , breath)

# f string method
print(f"The length of my rectangle is {length} and breadth is {breath}")

# .format method
print("The length of my rectangle is {} and breadth is {}".format(length,breath))
```

--To print more details of print

```
pip install loguru
from loguru import logger
```

logger.info("The length of my rectangle is {} and breadth is {}".format(length,breath))

Operators in Python

An operator in Python is a special symbol or keyword that performs an operation on one or more operands (values or variables).

```
x+y
x*y
x/y
x-y
x%y
```

Full list can be checked here :

https://docs.python.org/3/library/operator.html

Area = l * b

Print(name + length)

a="2"

b="3"

print(a+b)

Explicit type conversion

print(int(a)+int(b))

print(str(length)+str(breadth))

Implicit conversion

a=1.1

b=2

print(a+b)

--perimeter , --bodmas rule

p = 2*(l+b)

input() in Python

The input() function is used to **take input from the user** during program execution.

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

- The text inside the quotes is shown as a **prompt**.
- The user types something and presses **Enter**.
- The input is stored as a string (always!).

Find area of plot by taking input from user for length and breadth

⚠ Important:

Even if the user enters a number, input() returns it as a string. You need to convert it:

age = input("Enter your age: ")
print(age + 5) # ✗ Error: can't add string and int
☑ Fix:

age = int(input("Enter your age: "))
print(age + 5)

if else in python

The if...else statement is used for decision making in Python. It lets your program choose between different paths based on conditions.

```
if condition:
    # code if condition is True
elif another_condition:
    # code if another condition is True
else:
    # code if all conditions are False
```

Comparison operators need to be used in If condition like > ,== ,<

```
score = int(input("Enter your score: "))

if score >= 90:
    print("Grade: A")
elif score >= 75:
    print("Grade: B")
```

```
elif score >= 60:
    print("Grade: C")
else:
    print("Grade: F")
```

🧠 Remember:

- Conditions are checked **top to bottom**.
- Only the **first True** block runs.
- Indentation (spacing) is very important in Python.

Multiple conditions using and:

```
age = int(input("Enter your age: "))
has_id = input("Do you have an ID? (yes/no): ")

if age >= 18 and has_id.lower() == "yes":
    print("You are allowed to enter.")
else:
    print("Access denied.")
```

Multiple conditions using or:

```
city = input("Enter your city: ")

if city == "Dubai" or city == "Abu Dhabi":
    print("You are in the UAE.")
else:
    print("You are outside the UAE.")
```

NOT reverses the condition -> True to False and False to True

```
city = input("Enter your city: ")

if not (city == "Dubai" or city == "Abu Dhabi"):
    print("You are outside the UAE.")
else:
    print("You are in the UAE.")
```

Strings in Python

Strings in Python are sequences of characters enclosed in quotes, immutable and they are one of the most commonly used data types.

```
s1 = 'single quotes'
s2 = "double quotes"
s3 = """triple quotes
     for multi-line"""
```

1. Concatenation

```
first = "Data"
second = "Engineer"
full = first + " " + second
print(full)  # Output: Data Engineer
```

2. Length

```
s = "Hello"
print(len(s))  # Output: 5
```

3. Access Characters

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

4. Slicing

```
s = "DataEngineer"
print(s[0:4])     # Output: Data
print(s[4:])      # Output: Engineer
print(s[::-1])    # Output: reenignEataD (reversed)
```

name = 'Ankit'

name[startindex : endindex : stepsize]

For reverse slicing give negative value of stepsize -> negative indexing

5. String methods

```
s = "  hello world  "

print(s.upper())       # HELLO WORLD
print(s.strip())       # hello world
print(s.replace("world", "Python"))  # hello Python
print(s.startswith("h"))  # False (due to leading spaces)
print("world" in s)    # True
```

Strings are immutable

Name*2 ->

Questions : **Check if the first half of the string is the reverse of the second half.**

```
s = "abccba"
# Output: True
# First half = "abc", Second half = "cba" → reverse of "abc"
```

Number functions

** for power

`Pow()`

Abs

Round

Import math module and use math functions

Function	Description
math.sqrt(x)	Square root of x
math.ceil(x)	Rounds **up** to the nearest integer
math.floor(x)	Rounds **down** to the nearest integer
math.trunc(x)	Truncates decimal part (like int(x))
math.log(x)	Natural log (base e)
math.log10(x)	Logarithm base 10
math.exp(x)	e raised to the power of x
math.fabs(x)	Absolute value as float
math.isfinite(x)	Checks if x is neither inf nor NaN
math.pi	The constant π ≈ 3.14159
math.e	The constant e ≈ 2.71828

List in Python

A list in Python is an ordered, mutable collection of items. Lists can contain elements of any data type—integers, strings, other lists, even functions.

☑ Creating a List

```
my_list = [1, 2, 3, 4]
mixed_list = [1, "two", 3.0, [4, 5]]
empty_list = []
```

🔁 Accessing Elements

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

print(nums[0])    # 10
print(nums[-1])   # 40

# slicing
print(nums[1:3])  # [20, 30]
```

Modifying a List

```
 # add items
nums.append(50)        # Add at end
nums.insert(1, 15)     # Insert at index 1

 # remove items
nums.remove(20)        # Removes first occurrence of 20
popped = nums.pop()    # Removes and returns last item
popped = nums.pop(i)    # Removes element at index i
del nums[0]            # Deletes item at index 0

 # updating the element
nums = [10, 20, 30, 40]
nums[1] = 99
print(nums)        # [10, 99, 30, 40]

nums[2:4] = [300, 400]
print(nums)        # [10, 99, 300, 400]
```

➕ Combining Lists

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

■ Multi-dimensional Lists (Nested Lists)

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

print(matrix[0][1])      # 2
matrix[1][2] = 60        # Update middle row, last element
print(matrix)
# Output: [[1, 2, 3], [4, 5, 60], [7, 8, 9]]
```

📊 Useful List Methods

```
list1 = [3, 1, 4, 2]

list1.sort()            # Sort ascending
list1.reverse()         # Reverse order
print(len(list1))       # Length of list
print(list1.count(3))   # Count occurrences of 3
print(list1.index(4))   # Find index of 4
```

◇ split() Method in Python

The split() method is used to **split a string into a list** based on a **separator** (default is space).

```
string.split(separator, maxsplit)
```

- separator → (Optional) Delimiter to split by (default is space).
- maxsplit → (Optional) Maximum number of splits.

```
text = "Data Engineering Rocks"
words = text.split()
print(words)
# Output: ['Data', 'Engineering', 'Rocks']

line = "apple,banana,cherry"
fruits = line.split(",")
print(fruits)
# Output: ['apple', 'banana', 'cherry']
```

Split method -> email

sort

Assigments : reverse indexing eg get something from a url

Get question name from coding problem url

Join Method

The .join() method in Python is used to combine elements of a list (or any iterable) into a single string, using a specified separator.

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

- separator: The string you want to insert between elements (like space, comma, newline, etc.)
- iterable: A list or tuple of strings to join

Example: Join list of words with a space

```
words = ["Data", "Engineering", "is", "awesome"]
sentence = " ".join(words)
print(sentence)
# Output: Data Engineering is awesome
```

Example: Join with comma

```
items = ["apple", "banana", "cherry"]
result = ", ".join(items)
print(result)
# Output: apple, banana, cherry
```

Join with newline (for multi-line string)

```
lines = ["Line 1", "Line 2", "Line 3"]
output = "\n".join(lines)
print(output)
# Output:
# Line 1
# Line 2
# Line 3
```

⚠ Important Notes:

- All items in the iterable must be strings. If not, convert them first:

Question: You are given a sentence with extra spaces between words.
Clean up the sentence so that:

- There's only one space between words.
- No leading or trailing spaces.

```
s = "   Data    Engineering   is   awesome   "

s= " ".join(s.split())
# Output: "Data Engineering is awesome"
```

for loop

A for loop in Python is used to iterate over a sequence (like a list, tuple, string, or range) and execute a block of code for each element.

```
for item in sequence:
    # do something with item
```

Example

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

Looping with range

```
for i in range(5):  # from 0 to 4
    print("i is", i)
```

Courses : ['sql','python']

Course 1 is sql

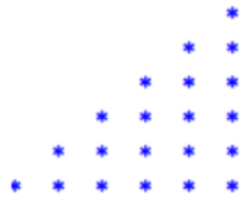
Course 2 is python

For course in courses

Range method

Create right angle triangle

Print all even number till 100



How many time SQL is there in this paragraph

SQL is a powerful language used to manage and query relational databases. With SQL you can easily retrieve specific data using SELECT statements, update records with UPDATE, or remove unwanted entries using DELETE. Learning sql is essential for data analysts, as most data stored in companies resides in SQL based systems. Advanced SQL techniques like window functions and common table expressions allow for more complex data manipulation. Whether you're building dashboards or running reports, SQL is the foundation of effective data analysis.

Insert 100 in a sorted list

```
a = [10,40,97,110,200]
```

After insert list should remain sorted: with and without lists insert function

Without insert

```
a.append(None)
            #(5,3,-1)
for j in range(len(a)-1,i,-1):
    if j>i :
        a[j]=a[j-1]

a[i]=100

print(a)
```

Talk about edge cases : if first element > 100 , or last element less than 100 or if list is empty

```
a = [10, 40, 97, 110, 200]
x = 100

if not a:
   a.append(x)
elif x <= a[0]:
   a.insert(0, x)
elif x >= a[-1]:
   a.append(x)
else:
   for i in range(len(a)):
      if x < a[i]:
         a.insert(i, x)
         break

print(a)
```

Loop on string:

You can **iterate over each character** in a string using a for loop.

☑ Example: Loop Through Characters

```
word = "Python"

for char in word:
    print(char)

#output
P
y
t
h
o
n
```

☑ Example: Count Vowels in a String

```
text = "Data Engineer"
vowels = "aeiouAEIOU"
count = 0

for char in text:
    if char in vowels:
        count += 1

print("Number of vowels:", count)
```

while loop

A while loop in Python repeatedly executes a block of code **as long as a given condition is True**.

```
while condition:
    # Code block to execute
```

Example

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

Output

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
```

🗀 Scenario: Wait until a file is uploaded (e.g., by another process or user)

```
import os
import time

file_path = "/path/to/your/file.csv"

print("⏳ Waiting for file to appear...")

while not os.path.exists(file_path):
    print("File not found. Checking again in 5 seconds...")
    time.sleep(5)

print("☑ File found! Proceeding with processing.")
```

🔍 Why while is ideal instead of For loop:

- You don't know **when** the file will appear.
- The loop should continue **as long as the file is missing**.
- You need to check the condition dynamically, not for a fixed number of times.

List Comprehension :

List comprehensions in Python are a concise way to create lists by applying an expression to each item in an iterable, optionally including a condition.

```
a = [10,40,97,110,200]
new_list= [number for number in a if number%2 ==0 ]
new_list = [ "Even" if number%2==0 else "odd" for number in a  ]
```

Dictionaries

A **dictionary** is an **unordered (ordered >3.7 version)**, **mutable** collection of **key-value pairs**. Keys must be unique and immutable (like strings, numbers, or tuples), values can be any type.

☑ Creating a Dictionary

```
ipl_teams = {
    "CSK": "Chennai Super Kings",
    "MI": "Mumbai Indians",
    "RCB": "Royal Challengers Bangalore",
    "KKR": "Kolkata Knight Riders",
    "SRH": "Sunrisers Hyderabad"
}

ipl_teams = {
    "CSK": {"captain": "MS Dhoni", "titles": 4},
    "MI": {"captain": "Rohit Sharma", "titles": 5},
    "RCB": {"captain": "Virat Kohli", "titles": 0}
}
```

🔍 Accessing Values by Key

```
print(ipl_teams["CSK"])    # Chennai Super Kings
print(ipl_teams["MI"])     # Mumbai Indians

print(ipl_teams.get("MI"))
print(ipl_teams.get("MI123"))
```

✏ Adding or Updating Items

```
ipl_teams["RR"] = "Rajasthan Royals"    # Add new team

ipl_teams["KKR"] = "Kolkata Knight Riders Updated"  # Update existing team name
```

➖ Removing Items

```
ipl_teams.pop("SRH")          # Remove Sunrisers Hyderabad
del ipl_teams["RR"]           # Delete Rajasthan Royals
```

📋 Dictionary Methods

```
print(ipl_teams.keys())         # dict_keys(['CSK', 'MI'])
print(ipl_teams.values())       # dict_values([{'captain': 'MS Dhoni', 'titles': 4},
{'captain': 'Rohit Sharma', 'titles': 5}])
```

```
print(ipl_teams.items())        # dict_items([('CSK', {...}), ('MI', {...})])
```

Looping Through a Dictionary

```
for team, info in ipl_teams.items():
    print(f"{team} captain is {info['captain']} and has won {info['titles']} titles")

# Output:
# CSK captain is MS Dhoni and has won 4 titles
# MI captain is Rohit Sharma and has won 5 titles
```

API response example

```
https://www.weatherapi.com/
{
  "location": {
    "name": "London",
    "region": "City of London, Greater London",
    "country": "United Kingdom",
    "lat": 51.52,
    "lon": -0.11,
    "tz_id": "Europe/London",
    "localtime_epoch": 1613896955,
    "localtime": "2021-02-21 8:42"
  },
  "current": {
    "last_updated_epoch": 1613896210,
    "last_updated": "2021-02-21 08:30",
    "temp_c": 11,
    "temp_f": 51.8,
    "is_day": 1,
    "condition": {
      "text": "Partly cloudy",
      "icon": "//cdn.weatherapi.com/weather/64x64/day/116.png",
      "code": 1003
    },
    "wind_mph": 3.8,
    "wind_kph": 6.1,
    "wind_degree": 220,
    "wind_dir": "SW",
    "pressure_mb": 1009,
    "pressure_in": 30.3,
    "precip_mm": 0.1,
    "precip_in": 0,
    "humidity": 82,
    "cloud": 75,
    "feelslike_c": 9.5,
```

```
    "feelslike_f": 49.2,
    "vis_km": 10,
    "vis_miles": 6,
    "uv": 1,
    "gust_mph": 10.5,
    "gust_kph": 16.9,
    "air_quality": {
      "co": 230.3,
      "no2": 13.5,
      "o3": 54.3,
      "so2": 7.9,
      "pm2_5": 8.6,
      "pm10": 11.3,
      "us-epa-index": 1,
      "gb-defra-index": 1
    }
  }
}

--single line
{"location":{"name":"London","region":"City of London, Greater
London","country":"United
Kingdom","lat":51.52,"lon":-0.11,"tz_id":"Europe/London","localtime_epoch":1613896955,
"localtime":"2021-02-21
8:42"},"current":{"last_updated_epoch":1613896210,"last_updated":"2021-02-21
08:30","temp_c":11,"temp_f":51.8,"is_day":1,"condition":{"text":"Partly
cloudy","icon":"//cdn.weatherapi.com/weather/64x64/day/116.png","code":1003},"wind_mph
":3.8,"wind_kph":6.1,"wind_degree":220,"wind_dir":"SW","pressure_mb":1009,"pressure_in
":30.3,"precip_mm":0.1,"precip_in":0,"humidity":82,"cloud":75,"feelslike_c":9.5,"feels
like_f":49.2,"vis_km":10,"vis_miles":6,"uv":1,"gust_mph":10.5,"gust_kph":16.9,"air_qua
lity":{"co":230.3,"no2":13.5,"o3":54.3,"so2":7.9,"pm2_5":8.6,"pm10":11.3,"us-epa-
index":1,"gb-defra-index":1}}}
```


1- Question : find number of time each character is present in "data engineering"

2- student_data = {
"Amit": {"Math": 87, "Science": 91, "English": 78},
"Riya": {"Math": 92, "Science": 85, "English": 88},
"Sneha": {"Math": 76, "Science": 81, "English": 79}

}

Take student name as input and produce below output. If student doesn't exists print " not present"

Output:

Marks of Riya:
Math: 92
Science: 85
English: 88
Average Marks: 88.33

```
student_data = {
   "Amit": {"Math": 87, "Science": 91, "English": 78},
   "Riya": {"Math": 92, "Science": 85, "English": 88},
   "Sneha": {"Math": 76, "Science": 81, "English": 79}
}

# Taking input from user
name = input("Enter student name: ")

# Check if the student exists in the dictionary
if name in student_data:
   print(f"Marks of {name}:")
   total = 0
   count = 0
   for subject, marks in student_data[name].items():
      print(f"{subject}: {marks}")
      total += marks
      count += 1
   average = total / count
   print(f"Average Marks: {round(average, 2)}")
else:
   print("Student not found")
```

Tuple in Python

A **tuple** is an **ordered**, **immutable** collection of elements. It's like a list, but you **can't change** it after creation.

```
# Basic tuple
colors = ("red", "green", "blue")

# Tuple with mixed data types
info = ("Ankit", 30, "Data Engineer")

# Single-element tuple (comma is important!)
single = ("hello",)

If you don't put comma then it will be treated as normal variable , int , str etc
```

Immutable

```
colors[0] = "yellow"  # ❌ This will give an error (tuples can't be changed)
```

📐 Tuple Packing and Unpacking

```
# Packing
person = ("Rohit", 35, "Mumbai")

# Unpacking
name, age, city = person
print(name)  # Rohit
print(city)  # Mumbai
```

🔁 Converting Between List and Tuple

```
list1 = [1, 2, 3]
tuple1 = tuple(list1)     # Convert list to tuple

tuple2 = (4, 5, 6)
list2 = list(tuple2)      # Convert tuple to list
```

1- Write a program to convert tuple of lists to flattened tuple

Input = ([1,2],[3,4,5,6,7],[2,3])

o/p = (1,2,3,4,5,6,7,2,3)

flattened = tuple(item for sublist in input_data for item in sublist)

input_data = ([1, 2], [3, 4, 5], [6, 7])

```
flattened_list = []

for lst in input_data:
    flattened_list.extend(lst)  # Directly extend without inner loop

flattened_tuple = tuple(flattened_list)

print(flattened_tuple)
```

2- Write a program to produce a tuple from 2 input tuples using power

Input1=(2,5,8,9)

Input2=(3,4,3,2)

o/p = (8,625,512,81)

```
input1 = (2, 5, 8, 9)
input2 = (3, 4, 3, 2)

result = []

for i in range(len(input1)):
    power = input1[i] ** input2[i]
    result.append(power)

# Convert list to tuple
output = tuple(result)

print(output)
```

Set in Python

A set is an unordered, mutable collection of unique elements.

☑ Creating a Set

```
my_set = {1, 2, 3, 4}
mixed_set = {1, "hello", 3.5}

# From a list (removes duplicates)
numbers = set([1, 2, 2, 3, 4, 4])
```

❌ Duplicates Are Removed Automatically

```
data = {1, 2, 2, 3}
print(data)  # Output: {1, 2, 3}
```

➕ Adding and Removing Elements

```
data = {1, 2}
data.add(3)

data.remove(2)      # Removes 2; error if not found
data.discard(5)     # No error if 5 is not found
```

🔁 Looping Through a Set

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

📊 Set Operations

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a.union(b))        # {1, 2, 3, 4, 5}
print(a.intersection(b)) # {3}
print(a.difference(b))   # {1, 2}
```

1- Given 2 lists find all the elements which are not common between them

# Convert to sets
set1 = set(list1)
set2 = set(list2)

```
    # Symmetric difference: elements in either set1 or set2 but not both
    result = list(set1.symmetric_difference(set2))
```

2- Find common elements among 3 lists (with and without sets)

```
list1 = [1, 2, 3, 4, 5, 5]

list2 = [3, 4, 5, 5, 6]

list3 = [5, 5, 6, 7, 8]
```

1-

```
# Convert to sets and take intersection
common = list(set(list1) & set(list2) & set(list3))
```

2-

```
common = []

for x in list1:
   if x in list2 and x in list3 and x not in common:
      common.append(x)
```

Functions in Python

In Python, functions are reusable blocks of code that perform a specific task. They help make your code modular, readable, and easier to maintain.

```
def function_name(parameters):
    # code block
    return result  # optional
```

Example 1: Function with Parameters and Return

```
def calculate_gst(price, rate):
    gst = price * rate / 100
    return price + gst

print(calculate_gst(1000, 18))  # Output: 1180.0
```

Example 2: Function with Default Argument

```
def calculate_gst(price, rate=18):
    gst = price * rate / 100
    return price + gst

print(calculate_gst(1000))  # Output: 1180.0
```

Example 3: Returning Multiple Values

```
def calculate_gst(price, rate):
    gst = price * rate / 100
    return gst , (price + gst)

gst,total_price = calculate_gst(1000, 18)
```

Problem: Find First Repeating Element in a List

Write a function that takes a list of numbers and returns the **first element** that repeats.
If no element repeats, return "No Repeats".

```
find_first_repeat([1, 2, 3, 2, 5, 6])  → 2
find_first_repeat([10, 20, 30, 40])   → "No Repeats"
```

```
def find_first_repeat(nums):
    seen = set()
    for num in nums:
        if num in seen:
```

```
            return num   # exits the loop early and function also over
        seen.add(num)
    return "No Repeats"
```

Problem: Check if a List is Strictly Increasing

Write a function that checks if a list of numbers is strictly increasing — meaning each number is greater than the one before it.
If the list is increasing, return "Increasing"
If not, return "Not Increasing"

```
def check_increasing(numbers):
    for i in range(1, len(numbers)):
        if numbers[i] <= numbers[i - 1]:
            return "Not Increasing"
    return "Increasing"
```

What are *args and **kwargs?

They allow a function to accept **a variable number of arguments**, making your functions flexible.

1. *args — Variable Number of Positional Arguments

- *args lets a function receive **any number of positional arguments** as a tuple.
- The name args is just a convention; you can use any name with a * before it.

Example 1:

```
def greet(*args):
    for name in args:
        print(f"Hello, {name}!")

greet("Ankit", "Sonia", "Ravi")
```

Example 2 : Summing Any Number of Numbers (*args) for a ecommerce cart

```
def sum_all(*numbers):
    total = 0
    for num in numbers:
        total += num
    return total

print(sum_all(10, 20, 30))        # Output: 60
print(sum_all(5, 15))             # Output: 20
print(sum_all(1, 2, 3, 4, 5, 6)) # Output: 21
```

2. **kwargs — Variable Number of Keyword Arguments

- **kwargs lets a function accept **any number of keyword (named) arguments** as a dictionary.
- The name kwargs is just a convention; you can use any name with ** before it.

Example 1:

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

print_info(name="Ankit", age=30, city="Bangalore")
```

Example 2: Building a User Profile (**kwargs)

```
def create_profile(**user_info):
    profile = {}
    for key, value in user_info.items():
        profile[key] = value
    return profile

user = create_profile(name="Ankit", age=30, city="Bangalore", profession="Teacher")
print(user)
```

Modules in python

- A **module** is a file containing Python definitions and statements (.py file).
- It can include functions, classes, variables, and runnable code.
- Modules help organize your code logically and reuse code across programs.

Why Use Modules?

- **Code reuse:** Write once, use many times.
- **Better organization:** Split big projects into smaller files.
- **Namespace management:** Avoid name clashes by grouping related code.
- Access to standard libraries: Python comes with many built-in modules (like math, os, random).

Packages in python

A package in Python is a way of organizing related modules together.

Example folder structure:

```
mypackage/
├── __init__.py
├── module1.py
└── module2.py
```

You can now import like this

```
from mypackage import module1
module1.my_function()
```

Notes :

__init__.py makes mypackage a **Python package**

Starting with Python 3.3, __init__.py is no longer strictly required, but still recommended for clarity.

What is __name__ in Python?

Every Python module (file) has a built-in variable called __name__.

- If the file is **run directly**, __name__ is set to "__main__".
- If the file is **imported** as a module in another script, __name__ is set to the **module's name** (i.e., the file name without .py).

It is a conditional statement used to control the execution of code.

```
if __name__ == "__main__":
    main()
```

This ensures that main() runs **only when the script is executed directly**, **not when it's imported**.

Example :

```
# sample.py

def greet():
    print("Hello!")

def main():
    print("Running the main function")

greet()

if __name__ == "__main__":
    main()
```

Run directly

```
python sample.py

Hello!
Running the main function
```

When imported

```
import sample

# output
```

```
Hello!
```

Example:

```
# math_utils.py
def addition(a, b):
    return a + b

def main():
    result = addition(2, 3)
    print(f"Sum: {result}")

if __name__ == "__main__":
    main()
```

Some important python inbuilt modules :

datetime

```
from datetime import datetime as dt
date = dt.now()
d = date(2024, 12, 25)
date = date.strftime("%Y-%m-%d")
print((date))
```

os

```
import os

print(os.getcwd())
print(os.listdir())
print(os.listdir("path"))
os.mkdir("new_folder")
```

Error handling ,working with Files and S3

What is Error Handling?

Sometimes, your program encounters **runtime errors** (like dividing by zero, missing files, or invalid input).
Error handling lets you **catch those errors gracefully** without crashing the program.

Basic Syntax: try - except

```
try:
    # risky code here
except SomeError:
    # what to do if error occurs
```

Example : catch all errors

```
try:
    something_risky()
except Exception as e:
    print("Something went wrong:", e)
```

Example : catch specific error

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("You cannot divide by zero!")

try:
    num = int("abc")
except ValueError:
    print("Invalid number!")

#multiple errors
try:
    num = int("abc")
except ValueError:
    print("Invalid number!")
except TypeError:
    print("Wrong type!")
```

else Block: Runs **only if no error occurs**.

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Error")
```

```
else:
    print("Success, result =", x)
```

finally Block: Runs always, whether error occurs or not. Great for clean-up.

```
try:
    file = open("myfile.txt", "r") # open a database connection
    content = file.read()
except FileNotFoundError:
    print("File not found")
finally:
    print("Closing file")
```

Working with files :

Opening a file:

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

Modes:

- 'r' – Read (default)
- 'w' – Write (creates file if not exists, overwrites if exists)
- 'a' – Append (adds to end of file)

Writing a file :

```
# Open file for writing (creates new or overwrites)
file = open("example.txt", "w")
file.write("Hello, this is a file.\n")
file.write("Writing to files is easy!\n")
file.close()
```

Reading from a file entire content :

```
# Open file for reading
file = open("example.txt", "r")
content = file.read()  # reads the entire file content
print(content)
file.close()
```

You can also read line-by-line:

```
file = open("example.txt", "r")
for line in file:
    print(line.strip())  # strip() removes newline characters
file.close()
```

Appending to a File

```
file = open("example.txt", "a")
file.write("Adding another line...\n")
file.close()
```

☑ Using with Statement (Best Practice)

Automatically closes the file when done:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)

with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())
with open("example.txt", "a") as file:
    file.write("Line added using with statement.\n")
```

☑ Reading a CSV File using open() and split()

```
# Open the CSV file
with open("data.csv", "r") as file:
    for line in file:
        # Remove newline character at end
        line = line.strip()

        # Split by comma to get list of values
        values = line.split(",")

        # Print the list
        print(values)
```

Using csv module:

The csv module in Python is used for reading from and writing to CSV (Comma-Separated Values) files in a clean and structured way. It's more robust than using split(','), especially when handling quoted values, embedded commas, or newlines inside cells.

Reading file

```
import csv

with open("data.csv", "r") as file:
    reader = csv.reader(file)
```

```
    for row in reader:
        print(row)

#skip header while reading
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)  # Skips header
    for row in reader:
        print(row)
```

Writing file:

```
data = [
    ["Name", "Age", "City"],
    ["Ankit", 30, "Bangalore"],
    ["Ravi", 25, "Mumbai"]
]

with open("output.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)  # write multiple rows
```

Working with s3:

boto3 library is used to interact with AWS services # pip install boto3

```
import boto3

s3 = boto3.client(
    "s3",
    aws_access_key_id="AKIA42TYBMSRX3M64RGK",
    aws_secret_access_key="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"
)
```

List buckets :

```
response = s3.list_buckets()
for bucket in response['Buckets']:
    print(bucket['Name'])
```

List files in a bucket:

```
bucket_name = "namastesql"
```

```
response = s3.list_objects_v2(Bucket=bucket_name)
for obj in response.get("Contents", []):
    print(obj["Key"])
```

Read content of a file

```
obj = s3.get_object(Bucket=bucket_name, Key="orders/orders_1.csv")
content = obj["Body"].read().decode("utf-8")
```

read content line by line

```
obj = s3.get_object(Bucket=bucket_name, Key="orders/orders_1.csv")
for line in obj["Body"].iter_lines():
    decoded_line = line.decode("utf-8")  # decode bytes to string
     print(decoded_line)
```

Upload and download a file

```
#Upload
s3.upload_file("files/orders.txt",bucket_name , "orders/20250605/orders.txt")
#download
s3.download_file(bucket_name, "orders/orders_1.csv", "orders_1.csv")
```

Configparser

it's a built-in Python library used to read and write configuration files, typically in .ini format.

What is configparser used for?

- Separating config values (like DB credentials, file paths, AWS keys) from your Python code
- Making your code more flexible and secure
- Useful when your app runs in different environments (local, test, prod)

Sample config.ini File

```
[aws]
access_key = ABCDEFG123456
secret_key = SECRET123456
region = us-east-1

[database]
host = localhost
port = 5432
user = admin
password = secret
```

Reading from config.ini Using configparser

```
import configparser

# Step 1: Create a ConfigParser object
config = configparser.ConfigParser()

# Step 2: Read the config file
config.read('config.ini')

# Step 3: Access values using section and key
aws_key = config['aws']['access_key']
aws_secret = config['aws']['secret_key']
region = config.get('aws', 'region')  # alternate way

db_host = config['database']['host']
db_port = config.getint('database', 'port')  # automatic int conversion

print("AWS Key:", aws_key)
print("Database Host:", db_host)
```

Checking Sections and Keys

```
print(config.sections())       # ['aws', 'database']
print(config.options('aws'))   # ['access_key', 'secret_key', 'region']
print(config.has_option('aws', 'access_key'))  # True
```

Install notebook

12 June 2025 12:39

```
pip install notebook
python -m notebook
```

Algorithms Overview

What is DSA?

DSA stands for Data Structures and Algorithms.

Data structures and algorithms are fundamental to efficient coding, impacting everything from code clarity and maintainability to performance and scalability. They provide a systematic approach to problem-solving, allowing programmers to choose optimal data storage and retrieval methods and to design efficient algorithms for various tasks. This leads to faster, more memory-efficient, and more robust software, crucial for real-world applications.

It is the foundation of writing efficient and scalable code, especially important in interviews, coding competitions, and real-world software development.

Concept	DSA Is About...
Data Structures	*Where* and *how* to store data
Algorithms	*How* to solve problems with that data

⏱️ What is Time Complexity?

Time Complexity tells you **how long** an algorithm takes to run **as the input size grows**.
We use **Big-O notation** (like O(n), O(log n), O(n^2)) to express this.

😵 Common Time Complexities:

Big-O	Description	Example Algorithm
O(1)	Constant time	Accessing an index in a list
O(n)	Linear time	Loop over a list (Linear Search)
O(log n)	Logarithmic time	Binary search
O(n^2)	Quadratic time	Nested loops (e.g. bubble sort)

O(1) Time Complexity :

O(1) means the operation takes the same amount of time, no matter how big the input size (n) is.

```
#example
accessing  a list element using index value
arr = [10,9,15,0,10]
arr[0]

# looking up a value into a set or dictionary also takes O(1) time.
```

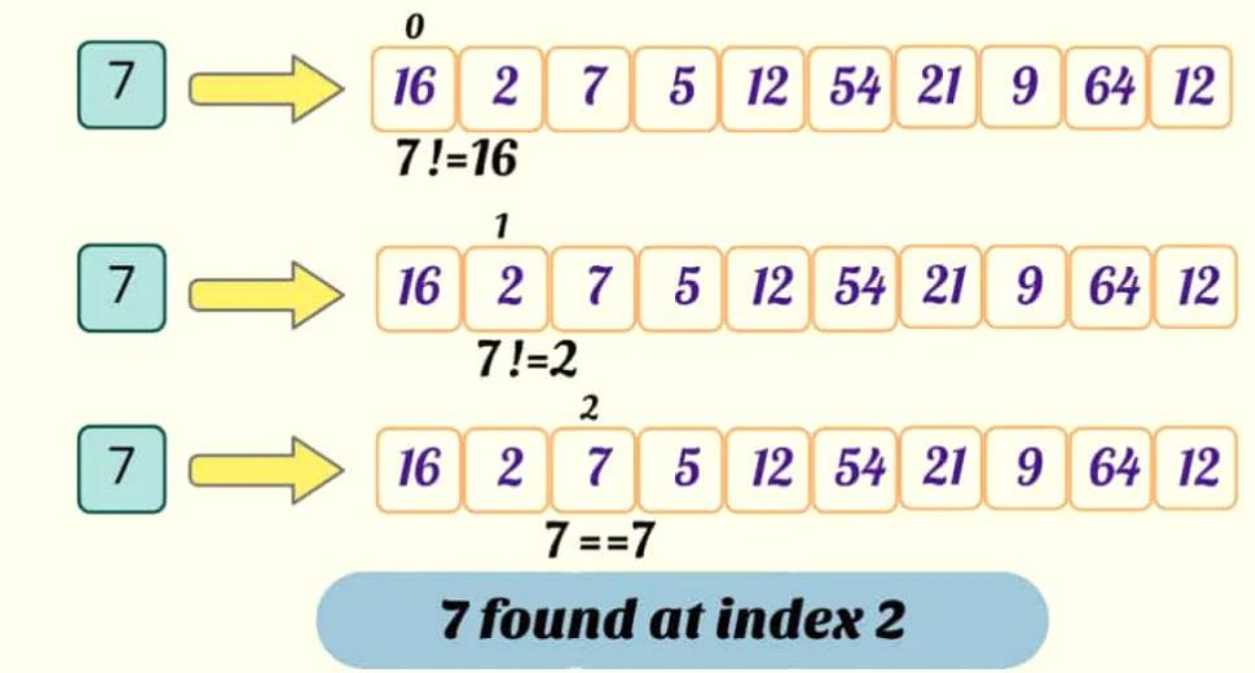
```python
import time

cities = set(range(100000000))
#cities = list(range(100000000))

start = time.time()
print(start)
found = 999995 in cities
end = time.time()
print(end)

print("Found:", found)
print("Lookup Time:", end - start)

----------------
valid_customers = {"cust_101": "Alice", "cust_102": "Bob"}

if "cust_102" in valid_customers:
    print(valid_customers["cust_102"])
```

O(n)

O(n) means the time taken by the algorithm grows linearly with the size of the input. If the input doubles, the time it takes also doubles.

e.g. Linear Search

Linear search is the most basic type of search that is performed. It is also called the sequential search. In this search, we check each element in the given list one by one until a match is found.



```python
# find index of a target element in a list
def linear_search(a,target):
    for i in range(len(a)):
        if a[i] == target:
            return i
```

Time Complexity : O(n)

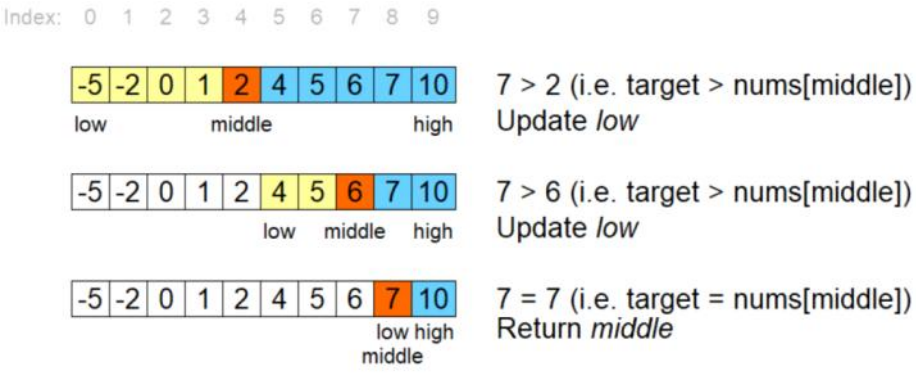
O(log n)

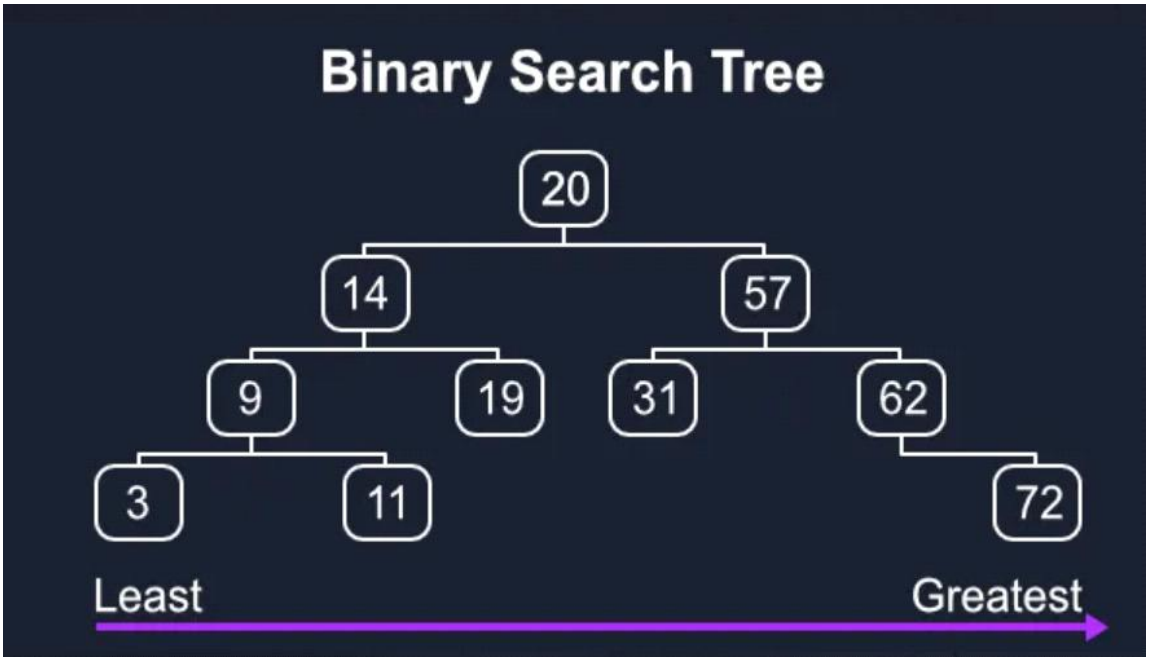
O(log n) means the time grows logarithmically - as the input size increases, the number of operations increases very slowly.

e.g. Binary Search

Binary search is an efficient algorithm for finding an element in a SORTED list. If the element is found, we return the index of the element in the list.

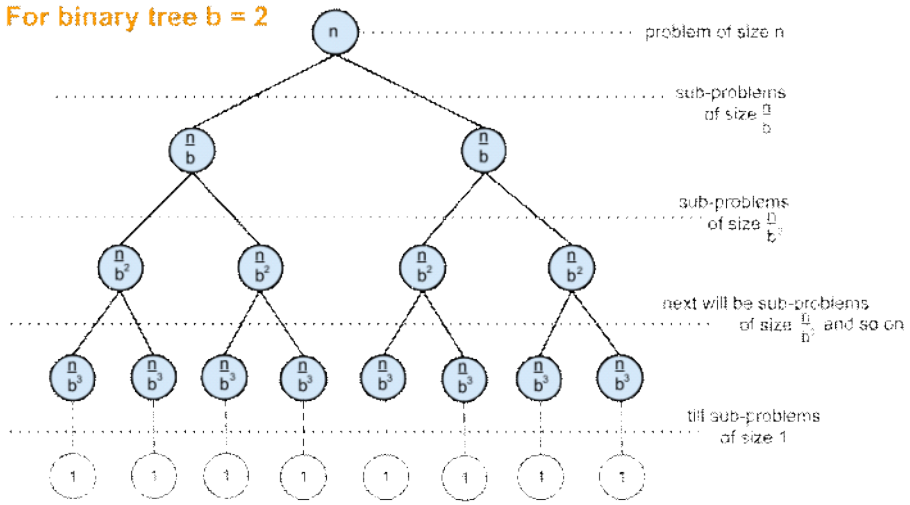
Target element = 7





```python
def binary_search(a, target):
    left = 0
    right = len(a)-1 #10
    while left <= right:
        print("loop")
        mid = (left+right)//2
        if a[mid] == target:
            return mid
        elif a[mid] < target :
            left=mid+1
        else :
            right = mid-1
```

Time complexity : O(log n)



The height of the above tree is answer to the following question: How many times we divide problem of size n by b until we get down to problem of size 1?

The other way of asking same question:

when $\frac{n}{b^x} = 1$ *[for binary tree b = 2]*

i.e. $n = b^x$ which is $\log_b n$ *[by definition of logarithm]*

O(n²) :

The time taken by the algorithm increases **quadratically** with the input size.

If you double the input, the time goes up by 4x.

e.g. 1 Print the below pattern :

For given input n, if n=5 print

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

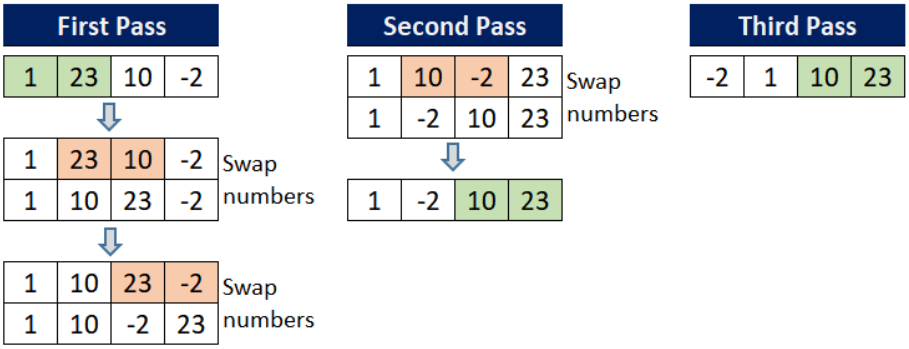
1 2 3 4 5

1 2 3 4 5

```python
n = 4
for i in range(n):
    for j in range(1, n + 1):
        print(j, end=" ")
    print()
```

e.g. 2 Bubble Sort

Bubble sort is the simplest sorting algorithm. The Bubble sort is based on the idea that every adjacent elements are compared and swapped if found in wrong order.



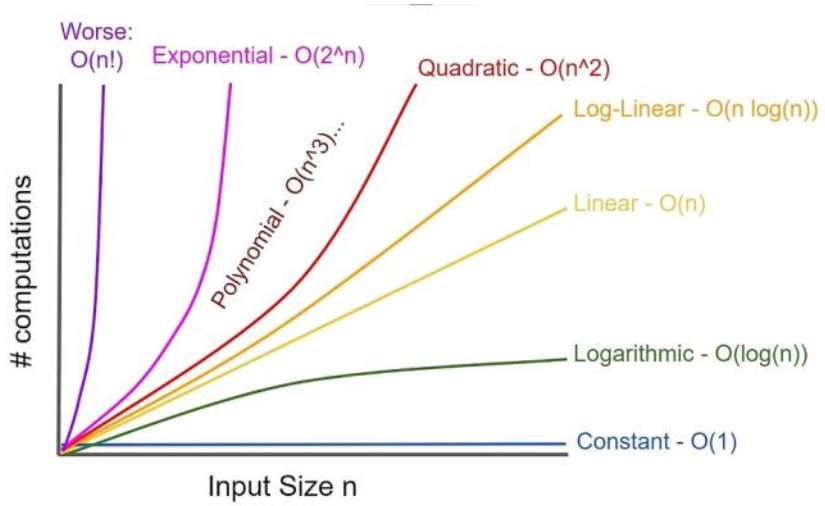

```python
def bubble_sort(a):
    n=len(a)
    for i in range(n):
        for j in range(n-1-i):
            if a[j] > a[j+1]:
                a[j],a[j+1]=a[j+1],a[j]
    return a
```

Time Complexity:

The time complexity of bubble sort is *O(N²)* in all cases even if the whole array is sorted because the entire array need to be iterated for every element and it contains two nested loops.

Time Comparison Table:

Input Size (n)	O(1)	O(log n)	O(n)	O(n²)
10	1	3.3	10	100
1000	1	10	1000	1,000,000
1,000,000	1	20	1,000,000	10^{12}



💾 What is Space Complexity?

Space Complexity tells you **how much extra memory** your algorithm uses **as input grows**.

Example :

```
def make_copy(arr):
    return arr.copy()
```

If arr has 100 items → new list of 100 items
☑ Space Complexity = **O(n)**

But:

```
def print_first(arr):
    print(arr[0])
```

Only one variable is used

☑ Space Complexity = O(1) (constant space)

🧠 Summary

Type	Measures	Goal
Time Complexity	Speed of algorithm	Make it faster
Space Complexity	Memory used by algorithm	Use less extra memory

OOPs in Python

OOPs in Python refers to Object-Oriented Programming - a programming paradigm that organizes code using classes and objects, making it more reusable, modular, and easier to maintain.

What is class

A class in Python is like a blueprint or template for creating objects. It defines the structure and behaviour (attributes and methods) that the objects created from it will have.

Think of it like this:

- **Class** = Blueprint of a car
- Object = Actual car built using that blueprint

Defining a class:

```
class Car:
    def __init__(self,make,model,year):
        self.make=make
        self.model=model
        self.year=year
        self.speed=0
```

__init__ in Python is called a constructor.

It is a **special method** that automatically runs **when an object is created from a class**. Its main job is to initialize (set up) the object with default or user-provided values.

What is object

An **object** in Python is a **real instance** of a class - it holds actual data and can perform actions (via methods) defined by the class.

```
car1=Car("mahindra","xuv300",2020)
car2=Car("Maruthi","Swift",2021)
```

Methods of a Class in Python

A method is a function that is defined inside a class and is used to define the behaviour of the objects created from that class.

```
class Car:
    color='Red'
    def __init__(self,make,model,year):
```

```
        self.make=make
        self.model=model
        self.year=year
        self.speed=0

    def accelerate(self,increment):
        self.speed=self.speed + increment

    def stop(self):
        self.speed=0
        print("car is stopped")
```

NumPy and Pandas

🧮 What is NumPy?

NumPy (short for **Numerical Python**) is a Python library used for **fast mathematical operations** on large arrays and matrices.
It's the **foundation for data science and machine learning** in Python — Pandas, Scikit-learn, TensorFlow, and many others use NumPy internally.

🔧 Why Use NumPy?

Because it's:
☑ **Faster** than Python lists
☑ **Memory-efficient**
☑ Built-in support for **matrix math**, **statistics**, **linear algebra**, **random numbers**, and more

Install pandas and notebook

```
pip install pandas
pip install notebook
python -m notebook
```

🐼 What is Pandas in Python?

Pandas is a powerful Python library used for data manipulation, analysis, and cleaning. It's especially useful when you're working with tabular data (like Excel or SQL tables).

You can:

- Load and inspect data
- Clean it
- Filter and manipulate it
- Analyse and aggregate it

```
data = {
    "name": ["Sumit", "Rahul", "Ankit"],
    "age": [25, 30, 35]
}

df = pd.DataFrame(data)
print(df)
```

📥 Reading a CSV file

```
import pandas as pd

df = pd.read_csv("orders.csv")
print(df.head())   # shows first 5 rows
type(df)
```

🔍 Exploring the DataFrame

```
df.head()          # First 5 rows
df.tail()          # Last 5 rows
df.shape           # (rows, columns)
df.columns         # List of column names
df.info()          # Data types and nulls
df.describe()      # Summary stats for numeric columns
df.dtypes          # datatypes
```

Rename column names:

```
#get list of columns and rename
df.columns
df.rename(columns = {'order_id':'id', 'ship_mode':'shipping_mode'},inplace=True)
df.columns = [list of columns]

#get list of indexes
df.index
```

Select columns

```
#single column
df["order_id"]
df.order_id
#multiple columns
df[["order_id", "customer_id"]]
```

Select rows and columns using iloc and loc

```
# by index position
df.iloc[0]        # First row
df.iloc[0:3]      # First 3 rows
df.iloc[0:3,0:2]  # First 3 rows, and first 2 columns
df.iloc[[0,2],[0,2]] #list of rows and column index

# rows by label/index name
df.loc[:, 'order_id']         # single column
df.loc[0, ['order_id','order_date']]         # Row with index = 0
df.loc[0:3,'order_id':'category']       # Same as iloc in default indexed DataFrames

# set a column as index
```

```
df.set_index('order_id',inplace=True)
df.loc["CA-2020-152156"]

# reset index
df.reset_index(inplace=True)
```

sorting dataframe

```
#sort based on index
df.sort_index(ascending=False)

#sort by a column
df.sort_values("sales")
# descending order
df.sort_values("sales", ascending=False)

#sort by multiple columns
df.sort_values(by=["category", "sales"], ascending=[True, False])
df.sort_values(by=["category", "sales"], ascending=[True, False], inplace=True)

#after sorting reset index
df.reset_index(drop=True) #drop=True to drop old index
```

Conditional filtering

```
fc = (df["category"]=="Furniture")
df[fc]

df[df["category"] == "Furniture"]
df[df["sales"] > 1000]
df[(df["category"] == "Technology") & (df["sales"] > 500)]
df[(df["category"] == "Technology") | (df["sales"] > 500)]

# Works with loc as well
df.loc[df["category"] == "Technology","order_id"]
df.loc[df["category"] == "Technology",["order_id","category"]]

# inverse a condition
fci = ~(df['profit']<0)
fci
```

Adding, Updating & Deleting Columns

```
#add new column
df["country"] = "USA"
df["unit_price"] = df["sales"]/df["quantity"]
```

```
df["region_category"] = df["region"] + df["category"]

# update a column
df["country"] = "India"
df["sales"] = df["sales"]*1.1

#update with condition
df.loc[df["profit"]<0 , "profit_flag"]=0

# add a conditional column
df.loc[df["profit"]>0 , "profit_flag"]=1
df.loc[df["profit"]<=0 , "profit_flag"]=0
df["profit_flag"] = df["profit_flag"].astype("int")

# drop column
df.drop(columns = "unit_price")

#drop rows
df.drop(index = [0,1,2])
```

Using apply to transform data

```
#inbuilt len function
df["category"].apply(len)

# user define function
def profit_loss(profit):
    if profit>0:
        return "profit"
    else:
        return "loss"

df["profit"].apply(profit_loss)
```

applying string functions

```
df["category"].str.lower()
df["order_id"].str.startswith("US")
```

Handling Missing Values in Pandas and drop duplicates

```
#check missing values
df.isnull()             # returns True/False for each cell
df.isnull().sum()       # total missing per column
df.info()               # quick view of non-null counts

# drop rows with missing values
```

```
df.dropna(inplace=True)              # removes rows with any missing value
df.dropna(subset=["sales"])   # removes rows where sales is missing

# fill missng values entire dataframe
df.fillna("Unknown", inplace=True)

# specific column
df["category"].fillna("Unknown", inplace=True)

# fill with mean/meadian
df["sales"].fillna(df["sales"].mean(), inplace=True)

# fill forward, fill backward
df.fillna(method="ffill", inplace=True)  # forward fill
df.fillna(method="bfill", inplace=True)  # backward fill



# Sometimes "NA", "?" or "missing" might be in the file. Replace them with nan:
import numpy as np
df.replace("missing", np.nan, inplace=True)

## drop duplicates
# remove complete duplicate rows , Keeps the first occurrence by default
df.drop_duplicates(inplace=True)

# keep last
df.drop_duplicates(inplace=True, keep = "last")

## drop based on specific columns
df.drop_duplicates(subset="customer_id", inplace=True)

## based on 2 columns
df.drop_duplicates(subset=["customer_id", "order_date"], inplace=True)

## just finding duplicates
df[df.duplicated(subset="customer_id", keep=False)]
```

Working with Dates & Times in Pandas

```
# Convert column to datetime (Pandas stores dates in a special format)
# default format is %Y-%m-%d
df["order_date"] = pd.to_datetime(df["order_date"] , format = "%d-%m-%Y")
df["ship_date"] = pd.to_datetime(df["ship_date"] , format = "%d-%m-%Y")

# extract parts of date
df["year"] = df["order_date"].dt.year
df["month"] = df["order_date"].dt.month
df["day"] = df["order_date"].dt.day
```

```python
df["weekday"] = df["order_date"].dt.day_name()

# filter by date
df[df["order_date"] >= "2023-01-01"]

#sort by date
df.sort_values("order_date", inplace=True)

# diff between 2 dates
df["transit_days"] = (df["ship_date"] - df["order_date"]).dt.days

# Set date as index (for time series) and do time series aggregations
df.set_index("order_date", inplace=True)
df.resample("M")["sales","profit"].sum()
```

Aggregation with pandas

```python
# simple aggregation on entire table
df["sales"].sum()      # Total sales
df["sales"].mean()     # Average sale amount
df["sales"].max()      # Highest order
df["sales"].min()      # Lowest order
df["sales"].count()    # Number of non-null orders
df["region"].nunique()  # get unique values of a column
df['category'].unique() # get distinct values of a column

# category wise sales
fc = (df['category'] == 'Technology')
df.loc[fc ,'sales'].sum()

# use group by to get grouped data at once
grouped_category = df.groupby("category")
type(grouped_category)
grouped_category.get_group("Technology")

# do aggregation along with group by
df.groupby("category")["sales"].sum()
df.groupby("category")["order_id"].count()

# group by multiple columns
df.groupby(["region", "category"])["sales"].sum()

# multiple aggregation at once
df.groupby("category")["sales"].agg(["min", "max"])


# aggregation on multiple columns
df.groupby('category').agg({'sales':'sum','profit':'sum'})
```

```
## By default, groupby() keeps the grouped column as index. To make it a normal column:
df.groupby("category")["sales"].sum().reset_index()
df.groupby('category', as_index=False)["sales"].sum()

## shortcut to get count of each value
df["category"].value_counts()
```

pivot table to summarize the data

```
pd.pivot_table(
    df,
    index="region",
    columns="category",
    values="sales",
    aggfunc="sum",
    fill_value=0
)
```

Joining or merging the data frames

```
df_orders = pd.read_csv("orders.csv")
df_returns = pd.read_csv("returns.csv")

# when the join column name is same in both dataframes
df_merged = pd.merge(left = df_orders, right= df_returns, how = 'left',on = "order_id")

# when the join column name is different in both dataframes
df_merged = pd.merge(left = df_orders , right= df_returns , how = 'left' , left_on =
"order_id", right_on = "return_order_id")

# how -> inner , left , right , outer , cross
```

Combining the data frames (Union ALL)

```
emp1 = {"id":[1,2,3],
      "name":["Ankit","Sumit","Amit"],
      "salary":[100,200,300]}
emp2 = {"id":[4,5,6],
      "name":["Nachiket","Rahul","Vinay"],
      "salary":[400,500,600]}

df1 = pd.DataFrame(emp1)
df2 = pd.DataFrame(emp2)
```

```
# union data frames
df = pd.concat([df1,df2])
df = pd.concat([df1,df2] , ignore_index= True)
```

shift dates (Similar to SQL lead and lag)

```
## shift the order_date
df["previous_order_date"] = df["order_date"].shift(1)

## sort the dataframe
df.sort_values(by= "order_date", inplace=True)
# use the shift to get the value from next row
df["previous_order_date"] = df["order_date"].shift(1)

## get next value within each group
df.sort_values(by=["customer_id", "order_date"], inplace=True)
df["previous_order_date"] = df.groupby("customer_id")["order_date"].shift(1)
```

Window Functions in Pandas

```
# 3-Day Moving Average
df.sort_values("order_date", inplace=True)
df["3_day_avg"] = df["sales"].rolling(window=3).mean()
df["3_day_avg"] = df.sort_values("order_date")["sales"].rolling(window=3).mean()

### within group
df.groupby("category")["sales"].rolling(window=3).sum().reset_index(level=0, drop=True)


# cumulative calculations
df["cum_sum"] = df["sales"].cumsum()
df["cum_max"] = df["sales"].cummax()
df["cum_min"] = df["sales"].cummin()

# ranking
df["sales_rank"] = df["sales"].rank(ascending=False)
# ranking within group
df["category_sales_rank"] = df.groupby("category")["sales"].rank(ascending=False)
```

Writing data from pandas to file

```
# export to csv
```

```
df.to_csv("output.csv", index=False)

#export to excel
df.to_excel("output.xlsx", index=False)

#export to json
df.to_json("data.json", orient="records", lines=True)

# writing data to multiple sheets of excel
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Score': [85, 92]})
df2 = pd.DataFrame({'Product': ['Pen', 'Pencil'], 'Price': [10, 5]})

# Write to multiple sheets
with pd.ExcelWriter('output.xlsx', engine='xlsxwriter') as writer:
    df1.to_excel(writer, sheet_name='Students', index=False)
    df2.to_excel(writer, sheet_name='Products', index=False)
```

Reading data from different file types

```
#csv
pd.read_csv("file.csv", delimiter=",")

#excel
df = pd.read_excel("sales.xlsx", sheet_name="Q1")

#json
df = pd.read_json("data.json")

#text file
df = pd.read_csv("data.txt", delimiter="|")

#parquet file
df = pd.read_parquet("data.parquet")

#clipbaord
df = pd.read_clipboard()
```

Working with database using cursor

```
# pip install mysql-connector-python
import mysql.connector
```

```
conn = mysql.connector.connect(
        host="127.0.0.1",        # or your DB host
        user="root",
        password="mysql123",
        database="sys"
    )

# cursor is a intermediater betwwen python and your database
cursor = conn.cursor()

# read data
cursor.execute("SELECT * FROM employees")

# fetch data one row at a time from cursor
row = cursor.fetchone()
print(row)
while True:
    row = cursor.fetchone()
    if row is None:
        break
    print(row)


# fetch all the rows together
data = cursor.fetchall()
for row in data:
     Print(row)

# write data


# # Define your INSERT query

# insert row with a hard coded value directly
insert_query = """
 INSERT INTO employees
 VALUES (108, 'Sunil','sunil@gmail.com')
 """

cursor.execute(insert_query)

conn.commit()

## insert single row using a tuple variable passed to execute
insert_query = """
 INSERT INTO employees
 VALUES (%s, %s, %s)
 """
data = (109, "Sohil", "sohil@gmail.com")
```

```
cursor.execute(insert_query, data)

## insert multiple rows together

data = [
    (1, "Alice", "Alice@gmail.com"),
    (2, "Bob", "bob@gmail.com"),
    (3, "Charlie", "cher@gmail.com"),
    (4, "David", "dav@gmail.com")
]
cursor.executemany(insert_query, data)

# update data
cursor.execute("UPDATE employees SET name = 'Ankit Bansal' WHERE id = 101")

cursor.close()
conn.commit()
conn.close()
```

Working with databases using pandas

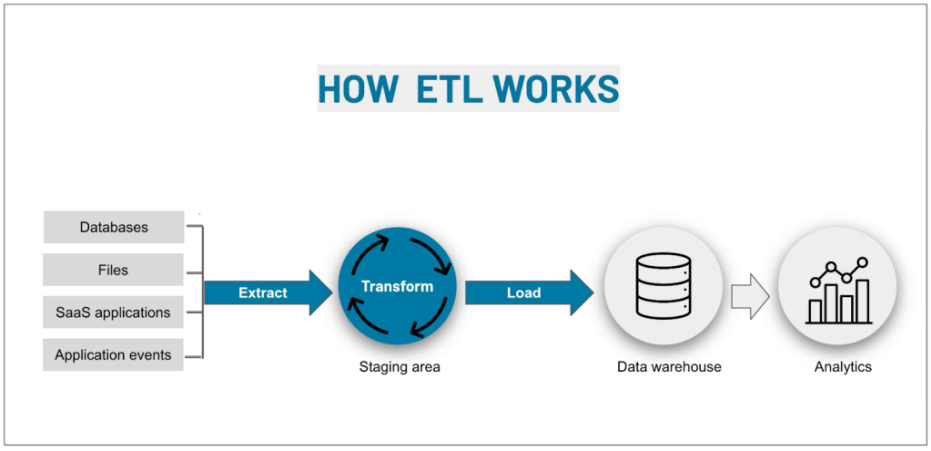
```
# mysql

## create connection
# pip install mysql-connector-python sqlalchemy pandas pymysql
from sqlalchemy import create_engine , text
import pandas as pd
engine = create_engine("mysql+pymysql://root:mysql123@127.0.0.1:3306/sys")

##read
query = "SELECT * FROM employees"
df = pd.read_sql(query, con=engine)

## write back
df.to_sql(name="employees_copy", con=engine, index=False, if_exists="replace")
# If_exists options:
'fail' (default): Error if table exists
'replace': Drop and recreate
'append': Add rows to existing table

# sql server (windows authentication)
from sqlalchemy import create_engine , text
import pandas as pd

# Replace server and database names accordingly
server = 'Ankit\SQLEXPRESS'
database = 'master'

# SQLAlchemy engine with Windows Authentication
```

```
engine = create_engine(
    f"mssql+pyodbc://@{server}/{database}?driver=ODBC+Driver+17
+for+SQL+Server&trusted_connection=yes"
)

#read data
df = pd.read_sql("SELECT * FROM orders", con=engine)

#write back
df.to_sql(name="orders_copy", con=engine, index=False, if_exists="replace")


## executing a statement
query = text("update employees set employee_name='Ankit' where employee_id = 101")

with engine.begin() as conn:
    conn.execute(query)
    conn.commit()
```

ETL : Extract , Transform and Load

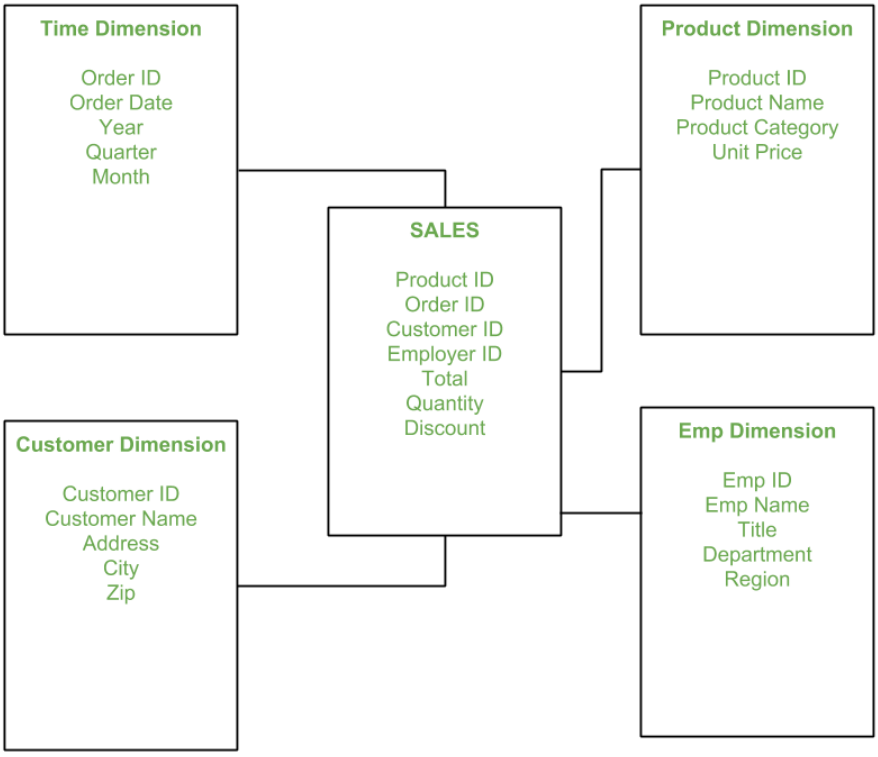
ETL is an automated process of integrating data from multiple sources into a central repository, such as a database or a data warehouse. It includes three steps which, if you look at their first letters, explain the ETL meaning well:

- Extraction
- Transformation or validation or data cleaning
- Loading



What is star schema

A star schema is a data modelling technique, primarily used in data warehousing, that simplifies the structure of a database for efficient querying and reporting. It consists of a central fact table surrounded by dimension tables, resembling a star shape when visualized. The fact table holds quantitative data (like sales figures or transaction counts), while dimension tables contain descriptive attributes (like product details, customer information, or time periods).


Time Dimension
Order ID
Order Date
Year
Quarter
Month
Product Dimension
Product ID
Product Name
Product Category
Unit Price
SALES
Product ID
Order ID
Customer ID
Employer ID
Total
Quantity
Discount
Customer Dimension
Customer ID
Customer Name
Address
City
Zip
Emp Dimension
Emp ID
Emp Name
Title
Department
Region

What is API

API stands for **Application Programming Interface**.
It allows two software systems to talk to each other.

Think of an API as a **restaurant menu**:

- The **menu (API)** tells you what you can order.
- You (the user) don't go into the kitchen — you just **request**.
- The **kitchen (server)** prepares and returns the food (data) to you.

Key Terms:

- **Endpoint**: A URL that performs a specific function (e.g., get weather info)
- **Request**: Your query to the API
- **Response**: What the API sends back (usually in JSON format)
- **Status code**: Tells if it worked (e.g., 200 OK, 404 Not Found)

Common packages to call API

```
import requests  # for making HTTP requests
import json      # for working with response data
```

Example : Get Weather by City (Open-Meteo, no API key)

```
## docs : https://open-meteo.com/en/docs
import requests
url = " https://api.open-meteo.com/v1/forecast"
params = {
    "latitude": 28.61,        # Delhi
    "longitude": 77.23,
    "current_weather": True
}

response = requests.get(url, params=params)
data = response.json()

print("Temperature:", data['current_weather']['temperature'], "°C")
```

```

## almost every API can be accessed using plain requests, because:

- At the core, APIs are just **HTTP endpoints** (URLs)
- You send GET, POST, etc. using requests
- You get back JSON or XML responses

Even if the API has its own official Python library, that library internally uses something like requests.
```

Example of airline and spotify APIs

Create package layer for AWS lambda

```
cd Desktop
mkdir lambda_package
cd lambda_package
mkdir python
cd python
pip install requests -t .
manually zip the file
```

Working with Snowflake

05 July 2025 10:29

Code:

```
#snowflake
#!pip install snowflake-connector-python pandas  snowflake-sqlalchemy

import snowflake.connector
import json
import pandas as pd
f = open("orders_ETL.json","r")
data = json.load(f)
orders_data=[]
for order in data:
        for product in order['products']:
                    row_orders = {
                        "order_id" : order["order_id"],
                        "order_date" : order["order_date"],
                        "total_amount" : order["total_amount"],
                        "customer_id" : order["customer"]["customer_id"],
                        "product_id" : product["product_id"],
                        "quantity" : product["quantity"]
                    }
                    orders_data.append(row_orders)

df_orders = pd.DataFrame(orders_data)
print(df_orders)

conn = snowflake.connector.connect(
    user='snowflakedemo',
    password='Snowflakedemo@123',
    account='RCRIYHI-EG46615',
    warehouse='COMPUTE_WH',
    database='LEARNSQL',
    schema='PUBLIC'
)

# Create a cursor
cur = conn.cursor()

# create table
cur.execute("CREATE OR REPLACE TABLE demo_table (order_id INT, order_date
date,total_amount decimal(5,1),customer_id int, product_id varchar(20),quantity int )")
# Clean up

# Insert data from a dataframe
insert_query = "INSERT INTO demo_table VALUES (%s, %s , %s, %s, %s, %s)"
```

```
cur.executemany(insert_query, df_orders.values.tolist())


cur.execute("SELECT * FROM demo_table")

# Fetch the data into a Pandas DataFrame
df = pd.DataFrame.from_records(
    cur.fetchall(),
    columns=[col[0] for col in cur.description]
)

cur.close()
conn.close()
```

Logging

What is Logging?

Logging is the process of recording messages during the execution of a program to track its behaviour, diagnose issues, and understand what's happening inside the code.
These messages called **logs** can include:

- Status updates (e.g. "Job started")
- Warnings (e.g. "API rate limit approaching")
- Errors (e.g. "File not found")
- Debug information (e.g. "user_id = 123")

Why is Logging Required?

1. ☑ **Debugging**
   Helps you trace issues without manually inserting print() statements everywhere.
2. ☑ **Visibility**
   Shows what's happening inside the application even after it crashes.
3. ☑ **Monitoring in Production**
   In real-world apps (like APIs, ETL pipelines, or Lambda functions), logs are the only way to know if things are working.
4. ☑ **Auditing**
   Useful to trace user actions, data changes, or errors for future reference.
5. ☑ **Better than print()**
   Logging allows log levels (info, error, debug), file output, rotation, timestamps — which print() doesn't provide.

```
#Instead of:
print("API called")
print("Error occurred")

# use
import logging

logging.basicConfig(level=logging.INFO , filename='logs.txt', format= '%(asctime)s
%(lineno)d', filemode='w')

logging.info("API called")
logging.error("Error occurred")
```

Logging Levels:

Level	Numeric value	What it means / When to use it
logging.**DEBUG**	10	Detailed information, typically only of interest to a developer trying to diagnose a problem.
logging.**INFO**	20	Confirmation that things are working as expected.
logging.**WARNING**	30	An indication that something unexpected happened, or that a problem might occur in the near future (e.g. 'disk space low'). The software is still working as expected.
logging.**ERROR**	40	Due to a more serious problem, the software has not been able to

		perform some function.
logging.**CRITICAL**	50	A serious error, indicating that the program itself may be unable to continue running.

Format options :

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into msg to produce message, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	%(asctime)s	Human-readable time when the LogRecord was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	%(created)f	Time when the LogRecord was created (as returned by time.time_ns() / 1e9).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la sys.exc_info) or, if no exception has occurred, None.
filename	%(filename)s	Filename portion of pathname.
funcName	%(funcName)s	Name of function containing the logging call.
levelname	%(levelname)s	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').
levelno	%(levelno)s	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).
lineno	%(lineno)d	Source line number where the logging call was issued (if available).
message	%(message)s	The logged message, computed as msg % args. This is set when Formatter.format() is invoked.
module	%(module)s	Module (name portion of filename).
msecs	%(msecs)d	Millisecond portion of the time when the LogRecord was created.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with args to produce message, or an arbitrary object (see Using arbitrary objects as messages).
name	%(name)s	Name of the logger used to log the call.
pathname	%(pathname)s	Full pathname of the source file where the logging call was issued (if available).
process	%(process)d	Process ID (if available).
processName	%(processName)s	Process name (if available).
relativeCreated	%(relativeCreated)d	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	%(thread)d	Thread ID (if available).
threadName	%(threadName)s	Thread name (if available).

taskNam e	%(taskName)s	asyncio.Task name (if available).

Create a custom logger

```
logger = logging.getLogger(__name__)
fh = logging.FileHandler("newlogs.txt")
f = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
fh.setFormatter(f)
logger.addHandler(fh)
logger.propagate = False
logger.info("from logger")
```

What is a Virtual Environment in Python?

A virtual environment is an isolated Python workspace that lets you install and manage packages independently for each project. It's like a separate Python universe for each project.

☑ Why is it useful?

1. Avoid conflicts between projects

Project A uses Pandas v1.5
Project B needs Pandas v2.1

You can't install both globally - but virtual environments solve this.

2. No need for admin rights

You can install packages locally without touching the system Python.

3. Cleaner deployments

You only install the packages you need for that specific project.

```
e virtual environment
python -m venv myvenv

# activate
myvenv\Scripts\activate

# Install packages inside myvenv

pip install pandas
pip install numpy

#deactivate
deactivate

# take snapshot of your env
pip freeze > requirements.txt

# recreate again from file
python -m venv myvenv
myvenv/Scripts/activate
pip install -r requirements.txt

# using in vs code
create virtal environment and use that (cahnge from right bottom or press ctrl+shift+p)
```

Multithreading

Multithreading means running **multiple threads (tasks) within a single process** concurrently allowing your program to do more than one thing at a time.

Each thread runs independently, but shares the same memory space.

When to Use Multithreading?

Use multithreading when your tasks are **I/O-bound** — for example:

- Reading files
- Making API calls
- Reading from S3 or a database
- Logging or writing to disk

⚠ Don't use it for **heavy CPU-bound** tasks — Python's **GIL (Global Interpreter Lock)** limits real parallelism for CPU-heavy work.

Let's understand with simple example

```
import threading
import time

def task(name):
    print(f"Starting task {name}")
    time.sleep(2)
    print(f"Finished task {name}")

start_time = time.time()  # Start timer

# # call the function without threading 3 times.

# t1 = task('A')
# t2 = task('B')
# t3 = task('C')

# Create 3 threads
t1 = threading.Thread(target=task, args=("A",))
t2 = threading.Thread(target=task, args=("B",))
t3 = threading.Thread(target=task, args=("C",))

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

#Wait for all threads to finish
t1.join()
t2.join()
```

```
t3.join()

end_time = time.time()  # End timer
total_time = end_time - start_time

print("All tasks completed", total_time)
```

What is GIL?

The **Global Interpreter Lock (GIL)** allows only **one thread** to execute **Python code** at a time per process.

That's a problem for CPU-heavy tasks. But when your thread hits an I/O wait — like:

```
# waiting for the API response
response = requests.get("https://api.example.com")
```

Python tells the OS: I'm waiting for a network response, I'm idle for now.

At that moment, the GIL is released, and another thread is allowed to run.

Real-world example: API Calls

Imagine calling 5 APIs one after the other:

```
❌ Without Threads: Takes ~10 seconds if each API takes 2 sec.
for url in urls:
    requests.get(url)  # Waits 1–2 sec each

☑ With Threads: Takes only ~2 seconds — all wait in parallel.

import threading
import time
import requests

# Dummy API that waits for 2 seconds
URL = "https://httpbin.org/delay/2"

def fetch_data(index):
    print(f"[Thread {index}] Starting request")
    response = requests.get(URL)
    print(f"[Thread {index}] Done with status code: {response.status_code}")

start_time = time.time()

threads = []

# Create 5 threads
for i in range(5):
    thread = threading.Thread(target=fetch_data, args=(i+1,))
```

```
    threads.append(thread)
    thread.start()

# Wait for all threads to finish
for thread in threads:
    thread.join()

end_time = time.time()
print(f"☑ All API calls completed in {end_time - start_time:.2f} seconds")

While thread 1 is waiting for API response, thread 2 is using the CPU to send another
request and so on
```

Analogy

Imagine 5 people ordering food at a counter.

- Without threads: Only 1 person allowed to stand in line. Others wait behind.
- With threads: All 5 place orders at once and **wait together**. As food comes, each collects it.

Waiting ≠ CPU work, so let threads wait while others do their thing.

Event pipeline

Lambda function

```
import json
import boto3
import io
from datetime import datetime
import pandas as pd
from io import StringIO


def flatten(data):
    orders_data=[]
    for order in data:
        for product in order['products']:
                    row_orders = {
                        "order_id" : order["order_id"],
                        "order_date" : order["order_date"],
                        "total_amount" : order["total_amount"],
                        "customer_id" : order["customer"]["customer_id"],
                        "customer_name" : order["customer"]["name"],
                        "email" : order["customer"]["email"],
                        "address" : order["customer"]["address"],
                        "product_id" : product["product_id"],
                        "product_name" : product["name"],
                        "category" : product["category"],
                        "price" : product["price"],
                        "quantity" : product["quantity"]
                    }
                    orders_data.append(row_orders)

    df_orders = pd.DataFrame(orders_data)
    return df_orders

def lambda_handler(event, context):
    print(event)
    # TODO implement
    bucket_name='namastesql'
    #key='orders_json/orders_ETL.json'
    key=event['Records'][0]['s3']['object']['key']
    # Create S3 client
    s3 = boto3.client('s3')

    # Get the object from S3
    response = s3.get_object(Bucket=bucket_name, Key=key)
```

```
    # Read and parse JSON content
    content = response['Body'].read().decode('utf-8')
    data = json.loads(content)
    df = flatten(data)

    # Convert DataFrame to CSV string in memory
    #csv_buffer = StringIO()
    #df.to_csv(csv_buffer, index=False)

     # in-memory binary buffer to hold data like a file, but without writing to disk.
    parquet_buffer = io.BytesIO()
    df.to_parquet(parquet_buffer, index=False, engine='pyarrow')

    now = datetime.now()
    timestamp = now.strftime("%Y%m%d_%H%M%S")

    #key_staging=f'orders_csv/orders_ETL_{timestamp}.csv'
    key_staging=f'orders_parquet/orders_ETL_{timestamp}.parquet'

    #s3.put_object(Bucket=bucket_name, Key=key_staging, Body=csv_buffer.getvalue())
    s3.put_object(Bucket=bucket_name, Key=key_staging, Body=parquet_buffer.getvalue())

    # crawler_name = 'your-crawler-name'
    # glue = boto3.client('glue')
    # response = glue.start_crawler(Name='orders_details')

    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```