Python Cheatsheet for Basic Coding

1. Basic Syntax

# Comments # Single-line

""" Multi-line comment """

# Print

print("Hello, World!")

1. Variables & Data Types

a = 10 # int

b = 3.14 # float name = "Anjan" # string is\_valid = True # boolean

x = None # NoneType

1. User Input

name = input("Name: ") age = int(input("Age: "))

a, b = map(int, input().split())

arr = list(map(int, input().split()))

matrix = [list(map(int, input().split())) for \_ in range(n)]

1. Data Structures

# **List**

A **list** is a collection of **ordered, changeable (mutable)** items.  
You can store **any type of data**: integers, strings, floats, even other lists!

empty = []

nums = [1, 2, 3]

mixed = [1, "apple", 3.14, True]

nested = [[1, 2], [3, 4]]

**📌 Accessing Elements**

fruits = ['apple', 'banana', 'mango']

print(fruits[0]) # apple

print(fruits[-1]) # mango (last element)

**🔁 Looping through Lists**

for item in fruits:

print(item)

# Using index

for i in range(len(fruits)):

print(fruits[i])

**✏️ Modifying Lists**

fruits[1] = 'grape' # Replace banana with grape

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

| **Method** | **Description** |
| --- | --- |
| append(x) | Add item to end |
| insert(i, x) | Insert item at index i |
| extend(list2) | Add items from another list |
| remove(x) | Remove first occurrence of x |
| pop([i]) | Remove and return item at index i |
| clear() | Remove all items |
| index(x) | Return index of first occurrence |
| count(x) | Count how many times x appears |
| sort() | Sort list (ascending) |
| reverse() | Reverse list order |

matrix = [

[1, 2],

[3, 4],

[5, 6]

]

print(matrix[1][0]) # 3

Check Item in List

if 'apple' in fruits:

print("Yes")

# **Set**

A **set** is a **collection of unique, unordered elements** in Python.

**Why use a set?**

* Automatically removes **duplicates**
* Fast **membership testing** (like x in set)
* Supports set operations: union, intersection, difference, etc.

## **🔸 How to create a set**

# Using curly braces

my\_set = {1, 2, 3, 4}

# Using the set() constructor

another\_set = set([2, 3, 5, 6])

📌 Duplicates are automatically removed:

s = {1, 2, 2, 3}

print(s) # Output: {1, 2, 3}

**🔸 Basic operations**

s = {1, 2, 3}

s.add(4) # Add an element

s.remove(2) # Remove an element (error if not found)

s.discard(5) # Safe remove (no error if not found)

s.clear() # Remove all elements

**🔸 Looping through a set**

for item in {10, 20, 30}:

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

10

20

30

my\_set = {10, 20, 30, 40}

print(len(my\_set)) # Output: 4

**🔸 Set operations**

a = {1, 2, 3}

b = {3, 4, 5}

print(a | b) # Union: {1, 2, 3, 4, 5}

print(a & b) # Intersection: {3}

print(a - b) # Difference: {1, 2}

print(a ^ b) # Symmetric Difference: {1, 2, 4, 5}

**🔸 Check membership**

print(2 in a) # True

print(5 in a) # False

# Dictionary

person = {"name": "Anjan", "age": 25}

1. Control Flow

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

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

while count < 5: count += 1

1. Functions

def add(a, b): return a + b

1. Error Handling

try:

result = 10 / 0

except ZeroDivisionError: print("Error")

finally:

print("Done")

1. Useful Built-in Functions

len(), type(), str(), int(), float(), list(), dict(), set(), input(), print(), sum(), max(), min(), sorted()

1. String Operations

text = "Hello, world!"

**🔁 Accessing Characters**

text = "Python"

print(text[0]) # P

print(text[-1]) # n

**🔁 Looping through a String**

for ch in text:

print(ch)

**🧪 Common String Methods**

| **Method** | **Description** |
| --- | --- |
| lower() | Converts to lowercase |
| upper() | Converts to uppercase |
| capitalize() | First letter uppercase, rest lowercase |
| title() | Capitalizes each word |
| strip() | Removes leading/trailing spaces |
| lstrip() / rstrip() | Left/right strip |
| replace(a, b) | Replace a with b |
| split() | Splits string into list using space |
| split(',') | Split using comma (or other delimiter) |
| join(list) | Join list into string with separator |
| find(sub) | Index of first occurrence of substring |
| count(sub) | Count occurrences of substring |
| startswith(str) | Returns True if string starts with str |
| endswith(str) | Returns True if string ends with str |
| isdigit() | True if all characters are digits |
| isalpha() | True if all are letters |
| isalnum() | True if all are letters or digits |
| swapcase() | Swap upper and lower case |

s = " Hello World "

print(s.lower()) # " hello world "

print(s.strip()) # "Hello World"

print(s.replace("World", "Python")) # " Hello Python "

print(s.split()) # ['Hello', 'World']

**🔤 String Slicing**

s = "Python"

print(s[0:4]) # "Pyth"

print(s[:]) # Full string

print(s[::-1]) # Reversed: "nohtyP"

**🔁 String Concatenation**

a = "Hello"

b = "World"

print(a + " " + b) # "Hello World"

**🧠 Tip: Strings are Immutable**

You **can't change** a string directly

s = "hello"

s[0] = 'H' # ❌ Error

**🔧 String Formatting**

**1. f-string (Python 3.6+)**

name = "Anjan"

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

**What is an f-string?**

* Introduced in **Python 3.6+**
* Let you insert variables **directly into strings**
* You put an f before the opening quote
* Inside the string, any **expression inside {} is evaluate**

Output-🡪 "Hello, Anjan!"

**2. format()**

print("Hello, {}".format(name))

we know string are immutable. But we can change the character of string by following approach.

One solution is to convert the string to a list and then change the value

>>> string = "abracadabra"

>>> l = list(string)

>>> l[5] = 'k'

>>> string = ''.join(l)

>>> print string

abrackdabra

or

* Another approach is to slice the string and join it back.
* >>> string = string[:5] + "k" + string[6:]
* >>> print string
* abrackdabra

1. List Comprehension

squares = [x\*x for x in range(5)]

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

1. File Handling

with open("file.txt", "w") as f: f.write("Hello")

with open("file.txt", "r") as f: content = f.read()

Common Shortcuts

Reverse: arr[::-1] Sort: sorted(arr) Sum: sum(arr)

Min/Max: min(arr), max(arr)

**What is a Tuple in Python?**

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

✅ Ordered → Elements have a fixed position  
✅ Immutable → You **cannot change** items after creation

**✅ Tuple Syntax**

my\_tuple = (1, 2, 3)

💡 If you want a **single-item tuple**, add a comma:

one\_item = (5,) # ✅ Tuple

not\_a\_tuple = (5) # ❌ Just an integer

**📚 Creating Tuples**

empty = ()

t1 = (10, 20, 30)

t2 = ("apple", True, 3.14)

nested = ((1, 2), (3, 4))

**🎯 Accessing Tuple Items**

t = (10, 20, 30)

print(t[0]) # 10

print(t[-1]) # 30 (last item)

**🔁 Looping Through Tuples**

for item in t:

print(item)

**🧠 Tuple is Immutable**

t = (1, 2, 3)

t[0] = 10 # ❌ Error: 'tuple' object does not support item assignment

**📦 Tuple Methods**

Tuples don’t have as many methods as lists, because they can't be changed.

| **Method** | **Description** |
| --- | --- |
| count(x) | Count occurrences of x |
| index(x) | First index of value x |

t = (1, 2, 3, 2)

print(t.count(2)) # 2

print(t.index(3)) # 2

| **Feature** | **List ([])** | **Tuple (())** |
| --- | --- | --- |
| Mutable | ✅ Yes | ❌ No |
| Ordered | ✅ Yes | ✅ Yes |
| Performance | 🔁 Slower | 🚀 Faster |
| Use case | Changeable data | Fixed data |

**When to Use Tuples?**

* When you want to protect data from being changed.
* As dictionary keys (lists can't be used as keys).
* For returning multiple values from a function.

## 🧪 Tuple Unpacking

point = (4, 5)

x, y = point

print(x) # 4

print(y) # 5

For Loop

**🔸 Example 1: Loop over a list**

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

for fruit in fruits:

print(fruit)

apple

banana

cherry

**🔸 Example 2: Loop over a string**

name = "Anjan"

for letter in name:

print(letter)

A

n

j

a

n

**🔸 Example 3: Using range()**

for i in range(5):

print(i)

0

1

2

3

4

**🔹 range(start, stop, step)**

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

print(i)

1

3

5

7

9

**🔸**

**Example 4: Loop with enumerate() (get index + value)**

colors = ['red', 'green', 'blue']

for index, color in enumerate(colors):

print(index, color)

0 red

1 green

2 blue

**🔸 Example 5: Nested for loop**

for i in range(1, 4):

for j in range(1, 3):

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

i=1, j=1

i=1, j=2

i=2, j=1

i=2, j=2

i=3, j=1

i=3, j=2

Some Useful Code Snippet :

**score\_list = map(int, input().split())**

**Explanation:**

This line is taking input from the user and converting it into a sequence of integers.

**Step-by-step breakdown:**

1. **input()**
   * This function takes a line of input from the user as a **string**.
   * Example input: "5 3 6 6 2"
2. **.split()**
   * This splits the input string into a **list of substrings** using whitespace as the default delimiter.
   * Example: "5 3 6 6 2".split() → ['5', '3', '6', '6', '2']
3. **map(int, ...)**
   * The map() function applies the int function to **each item** in the list.
   * Converts each string in the list to an integer.
   * Example:  
     map(int, ['5', '3', '6', '6', '2']) → map object (an iterable of integers)
4. **Assignment to score\_list**
   * score\_list now holds a map object containing integers.
   * In the next line (score\_list = list(score\_list)), this is converted to a list of integers:

[5, 3, 6, 6, 2]

If you print(**score\_list) it will give obj reference. As map return object if you want to print values have to convert it into list. Print(List(score\_list)) // 5 3 6 6 2**

Nested list problem in hackerrank :

if \_\_name\_\_ == '\_\_main\_\_':

name\_list=[]

score\_list=[]

nested\_record\_list = []

for i in range(int(input())):

name = input()

score = float(input())

name\_list.append(name)

score\_list.append(score)

nested\_record\_list.append([name,score])

unique\_score\_list = list(set(score\_list))

unique\_score\_list.sort()

second\_lowest\_score = unique\_score\_list[1]

# out = [i[0] for i in nested\_record\_list if i[1] == second\_lowest\_score]

#

out=[]

for i in nested\_record\_list:

NAME = i[0]

SCORE = i[1]

if SCORE == second\_lowest\_score:

out.append(NAME)

out.sort()

for i in out:

print(i)

Dictonary Problem In hacker rank

if \_\_name\_\_ == '\_\_main\_\_':

    n = int(input())

    student\_marks = {} ## creating DICT

    for i in range(n):

        data= input()

        ##print(data)

        data=data.split()

        ##print(data)

        name = data[0]

        marks= data[1:]

        scores = []

        i=0

        for score in marks:

            scores.append(float(marks[i]))

            i=i+1

        # name, \*line = input().split()

        # scores = list(map(float, line))

        student\_marks[name] = scores ## PUT SCORE IN DICT

        ##print(student\_marks)

    query\_name = input()

    print(format(sum(student\_marks[query\_name])/len(student\_marks[query\_name]),".2f"))

https://www.hackerrank.com/challenges/swap-case/problem?isFullScreen=true

def swap\_case(s):

    s1=""

    for i in s:

        if i.islower():

          s1 += i.upper()

        elif i.isupper():

         s1 += i.lower()

        else:

         s1 += i

    return s1

if \_\_name\_\_ == '\_\_main\_\_':

    s = input()

    result = swap\_case(s)

    print(result)

https://www.hackerrank.com/challenges/python-string-split-and-join/problem?isFullScreen=true

def split\_and\_join(line):

    # write your code here

    #print(line)

    line=line.split()

    line="-".join(line)

    return line

if \_\_name\_\_ == '\_\_main\_\_':

    line = input()

    result = split\_and\_join(line)

    print(result)

https://www.hackerrank.com/challenges/find-a-string/problem?isFullScreen=true

def count\_substring(string, sub\_string):

     count=0

     for i in range(len(string)- len(sub\_string)+1):

        if string[i: i+ len(sub\_string)] == sub\_string:

            count+=1

     return count

if \_\_name\_\_ == '\_\_main\_\_':

    string = input().strip()

    sub\_string = input().strip()

    count = count\_substring(string, sub\_string)

    print(count)

**Numpy**

**Way to build numpy array:**

* 1. **From List we can build numpy array.**

import numpy as np

arr = np.array([1, 2, 3, 4]) // creating numpy array

print(arr) [1 2 3 4]

**🔹 Multidimensional array (Matrix):**

matrix = np.array([[1, 2], [3, 4]])

print(matrix)

[[1 2]

[3 4]]

**How to user input matrix** import numpy as np

row,col = map(int, input().split())

# print(row,col)

arr=np.array([list(map(int,input().split())) for \_ in range(row)])

print(arr)

 input().split() takes space-separated numbers as strings.

 map(int, ...) converts each to an integer.

 list(...) converts the mapped values into a list of integers.

 The outer list comprehension runs row1 times — so you get a list of lists (i.e., a 2D list).

 np.array(...) turns the 2D list into a proper NumPy 2D array.

**✅ Step 4: Array Properties**

a = np.array([[1, 2, 3], [4, 5, 6]])

print(a.shape) # (2, 3) → 2 rows, 3 columns

print(a.ndim) # 2 → 2D array

print(a.size) # 6 → total number of elements

print(a.dtype) # int32 or int64 depending on your system

Step 5: Creating Special Arrays

np.zeros((2, 3)) # 2x3 array of zeros

np.ones((3, 2)) # 3x2 array of ones

np.eye(3) # 3x3 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. ]

np.random.rand(2,3) 2×3 matrix of random numbers

arr.reshape(new\_shape)

Where new\_shape can be a tuple like (rows, columns).

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

reshaped = arr.reshape(2, 3)

print(reshaped)

Using -1 (automatic dimension)

arr = np.array([1, 2, 3, 4, 5, 6])

reshaped = arr.reshape(-1, 2) # Let NumPy figure out rows

print(reshaped)

[[1 2]

[3 4]

[5 6]]

-1 tells NumPy to automatically calculate that it must be 3 rows (since 6 ÷ 2 = 3).

arr.ravel() in NumPy — it's used to **flatten** multi-dimensional arrays into a **1D array**.

arr.ravel()

import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

flat = arr.ravel()

print(flat) [1 2 3 4 5 6]

**What is axis in NumPy?**

In NumPy, **axis tells the direction** along which a function is applied (like sum(), mean(), concatenate(), etc.).

**🧭 Think of axes like dimensions:**

* **Axis 0** → down the rows (**vertically**)
* **Axis 1** → across the columns (**horizontally**)

import numpy as np

arr = np.array([[1, 2, 3],

[4, 5, 6]])

This array has:

* **2 rows** → axis 0
* **3 columns** → axis 1

## Examples with Axis

### 🧮 1. Sum over rows (axis=0)

np.sum(arr, axis=0)

🟢 Output

[5 7 9] ← Column-wise sum: [1+4, 2+5, 3+6]

### 🧮 2. Sum over columns (axis=1)

python

CopyEdit

np.sum(arr, axis=1)

🟢 Output:

less

CopyEdit

[6 15] ← Row-wise sum: [1+2+3, 4+5+6]

## 🔄 Other examples using axis

### 🌟 Mean:

python

CopyEdit

np.mean(arr, axis=0) # mean of each column

np.mean(arr, axis=1) # mean of each row

import numpy as np

a = np.array([[1, 2],

[3, 4]])

b = np.array([[5, 6],

[7, 8]])

**concatenate((a, b), axis=0)**

➡️ This stacks the arrays **vertically** (row-wise) — i.e. adds b's rows **below** a.

result = np.concatenate((a, b), axis=0)

print(result)

[[1 2]

[3 4]

[5 6]

[7 8]]

**🔹 np.concatenate((a, b), axis=1)**

➡️ This stacks the arrays **horizontally** (column-wise) — i.e. adds b's columns **next to** a.

result = np.concatenate((a, b), axis=1)

print(result)

[[1 2 5 6]

[3 4 7 8]]

**✅ ar.flat in NumPy**

array.flat

 It returns a **1D iterator** over the array.

 Unlike ravel() or flatten(), it does **not return a new array**, just a way to **iterate** over the elements as if it's flat.

for i in ar.flat:

print(i)

1

2

3

4

print(ar.flat[2]) # 3rd element in flat order 3

**When to use .flat?**

* When you want to loop through all elements regardless of array shape
* When you want to access elements using a **single flat index**

np.argmax() returns the **index of the maximum value** in an array.

import numpy as np

arr = np.array([10, 20, 50, 30])

print(np.argmax(arr))

2

Opposite to argmin()

np.argsort() returns the **indices** that would **sort the array**.

**Python Pandas**

**What is Pandas?**

**Pandas** is a powerful Python library used for:

* Handling and analyzing structured data (like Excel, CSV, SQL)
* Performing data cleaning, manipulation, and exploration
* It provides two main data structures:
  + Series → 1D labeled array
  + DataFrame → 2D labeled table (like Excel)

# Create a simple DataFrame

data = {

'Name': ['Alice', 'Bob', 'Charlie'], //dictonary

'Age': [25, 30, 22],

'City': ['Pune', 'Mumbai', 'Delhi']

}

df = pd.DataFrame(data)

print(df)

**What is a DataFrame?**

A **DataFrame** in Pandas is a **2D labeled table** — just like an Excel sheet or SQL table.

Each DataFrame:

* Has **rows** and **columns**
* Has **labels** (called indexes for rows and column names)
* Is built from a dictionary, list of lists, or external files (CSV, Excel)

Creating a DataFrame

import pandas as pd

data = {

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 22],

'City': ['Pune', 'Mumbai', 'Delhi']

}

df = pd.DataFrame(data)

print(df)

**🧪 DataFrame Basics – Key Operations**

df.head() → View first few rows print(df.head())

**🔹 df.tail() → View last few rows print(df.tail())**

**df.shape → Get number of rows and columns print(df.shape) # Output: (3, 3)**

**df.columns → Get column names**

print(df.columns) # Index(['Name', 'Age', 'City'], dtype='object')

df.dtypes → Data types of columns df.dtypes → Data types of columns

**df.info() → Summary of DataFrame**

df.info()

**Accessing Data in a DataFrame**

Access a column print(df['Name']) # or df.Name

**Access a specific row:**

print(df.loc[0]) # By label (index)

print(df.iloc[0]) # By position

**Multiple columns:**

print(df[['Name', 'City']])

**Add a new column:**

df['Country'] = 'India'

**Example: Modify Data**

df['Age'] = df['Age'] + 1 # Increase all ages by 1

print(df)

| **Task** | **Code** |
| --- | --- |
| Create DataFrame | pd.DataFrame(data) |
| View top rows | df.head() |
| Access column | df['ColumnName'] |
| Add new column | df['NewCol'] = values |
| Get shape (rows, cols) | df.shape |
| Row by index | df.loc[0] or df.iloc[0] |

**Read and Write Excel and CSV Files**

**import pandas as pd**

**# -----------------------------**

**# READ CSV FILE**

**# -----------------------------**

**# Make sure 'students.csv' is in the same folder**

**df\_csv = pd.read\_csv('students.csv')**

**print("📥 CSV Data Read:")**

**print(df\_csv)**

**# -----------------------------**

**# WRITE TO CSV FILE**

**# -----------------------------**

**df\_csv.to\_csv('new\_students.csv', index=False)**

**print("\n✅ CSV file 'new\_students.csv' written successfully.")**

**# -----------------------------**

**# READ EXCEL FILE**

**# -----------------------------**

**# Make sure 'students.xlsx' exists**

**df\_excel = pd.read\_excel('students.xlsx')**

**print("\n📥 Excel Data Read:")**

**print(df\_excel)**

**# -----------------------------**

**# WRITE TO EXCEL FILE**

**# -----------------------------**

**df\_excel.to\_excel('new\_students.xlsx', index=False)**

**print("\n✅ Excel file 'new\_students.xlsx' written successfully.")**

**✅ Full Example: Handle NA Values – Part 1**

| **Task** | **Function** |
| --- | --- |
| **Check missing** | **df.isna()** |
| **Count missing per column** | **df.isna().sum()** |
| **Drop rows with NA** | **df.dropna()** |
| **Drop columns with NA** | **df.dropna(axis=1)** |
| **Fill NA with a value** | **df.fillna(value)** |

**import pandas as pd**

**import numpy as np**

**# Sample DataFrame with missing values (NaN)**

**data = {**

**'Name': ['Alice', 'Bob', 'Charlie', 'David'],**

**'Age': [25, np.nan, 22, np.nan],**

**'City': ['Pune', 'Mumbai', np.nan, 'Delhi']**

**}**

**df = pd.DataFrame(data)**

**print("🟡 Original DataFrame:")**

**print(df)**

**print('-' \* 50)**

**# 1️⃣ Detect Missing Values**

**print("🔍 df.isna(): Shows True where values are missing")**

**print(df.isna())**

**print('-' \* 50)**

**# 2️⃣ Count Missing Values in Each Column**

**print("📊 df.isna().sum(): Count of missing values per column")**

**print(df.isna().sum())**

**print('-' \* 50)**

**# 3️⃣ Drop Rows with Any Missing Values**

**print("❌ df.dropna(): Drop rows that contain any NaN")**

**df\_drop\_rows = df.dropna()**

**print(df\_drop\_rows)**

**print('-' \* 50)**

**# 4️⃣ Drop Columns with Any Missing Values**

**print("❌ df.dropna(axis=1): Drop columns that contain any NaN")**

**df\_drop\_cols = df.dropna(axis=1)**

**print(df\_drop\_cols)**

**print('-' \* 50)**

**# 5️⃣ Fill Missing Values with 0 (preview of Part 2)**

**print("🔁 df.fillna(0): Replace all NaN with 0")**

**df\_fill\_zero = df.fillna(0)**

**print(df\_fill\_zero)**

**Output**

**🟡 Original DataFrame:**

**Name Age City**

**0 Alice 25.0 Pune**

**1 Bob NaN Mumbai**

**2 Charlie 22.0 NaN**

**3 David NaN Delhi**

**--------------------------------------------------**

**🔍 df.isna(): Shows True where values are missing**

**Name Age City**

**0 False False False**

**1 False True False**

**2 False False True**

**3 False True False**

**--------------------------------------------------**

**📊 df.isna().sum(): Count of missing values per column**

**Name 0**

**Age 2**

**City 1**

**dtype: int64**

**--------------------------------------------------**

**❌ df.dropna(): Drop rows that contain any NaN**

**Name Age City**

**0 Alice 25.0 Pune**

**--------------------------------------------------**

**❌ df.dropna(axis=1): Drop columns that contain any NaN**

**Name**

**0 Alice**

**1 Bob**

**2 Charlie**

**3 David**

**--------------------------------------------------**

**🔁 df.fillna(0): Replace all NaN with 0**

**Name Age City**

**0 Alice 25.0 Pune**

**1 Bob 0.0 Mumbai**

**2 Charlie 22.0 0**

**3 David 0.0 Delhi**

**Handle NA Values – Part 2 with each operation and its full example**

**import pandas as pd**

**import numpy as np**

**# Sample data**

**data = {**

**'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],**

**'Age': [25, np.nan, 22, np.nan, 30],**

**'City': ['Pune', 'Mumbai', np.nan, 'Delhi', np.nan]**

**}**

**df = pd.DataFrame(data)**

**print("🔸 Original DataFrame:\n", df)**

**Fill with a constant value**

**df.fillna("Unknown")**

**Name Age City**

**0 Alice 25 Pune**

**1 Bob Unknown Mumbai**

**2 Charlie 22 Unknown**

**3 David Unknown Delhi**

**4 Eva 30 Unknown**

**. Forward fill (fill with previous value)**

**df.fillna(method='ffill')**

**Backward fill (method = 'bfill')**

**df.fillna(method='bfill')**

**✅ 4. Forward fill with limit = 1**

**df.fillna(method='ffill', limit=1)**

**Fill numeric column (Age) with mean**

**df['Age'].mean() # 25 + 22 + 30 = 77 / 3 = 25.666...**

**df['Age'] = df['Age'].fillna(df['Age'].mean())**

**Name Age City**

**0 Alice 25.000000 Pune**

**1 Bob 25.666667 Mumbai**

**2 Charlie 22.000000 NaN**

**3 David 25.666667 Delhi**

**4 Eva 30.000000 NaN**

**Fill text column (City) with mode**

**python**

**CopyEdit**

**df['City'].mode()[0] # Output: 'Delhi'**

**df['City'] = df['City'].fillna('Delhi')**

**🔹 Output:**

**markdown**

**CopyEdit**

**Name Age City**

**0 Alice 25.000000 Pune**

**1 Bob 25.666667 Mumbai**

**2 Charlie 22.000000 Delhi**

**3 David 25.666667 Delhi**

**4 Eva 30.000000 Delhi**

**Final Cleaned DataFrame (After All Imputations):**

**python**

**CopyEdit**

**df['Age'] = df['Age'].fillna(df['Age'].mean())**

**df['City'] = df['City'].fillna(df['City'].mode()[0])**

**🔹 Final Output:**

**markdown**

**CopyEdit**

**Name Age City**

**0 Alice 25.000000 Pune**

**1 Bob 25.666667 Mumbai**

**2 Charlie 22.000000 Delhi**

**3 David 25.666667 Delhi**

**4 Eva 30.000000 Delhi**

**What is groupby()?**

**groupby() in Pandas is used to:**

* **Split data into groups**
* **Apply a function (like sum, mean, count, etc.)**
* **Combine the results into a new DataFrame**

**This is similar to SQL's GROUP BY clause.**

**import pandas as pd**

**data = {**

**'Region': ['East', 'West', 'East', 'West', 'East', 'South'],**

**'Salesperson': ['Alice', 'Bob', 'Alice', 'Bob', 'Charlie', 'David'],**

**'Sales': [250, 300, 150, 200, 400, 500]**

**}**

**df = pd.DataFrame(data)**

**print("🔸 Original Data:\n", df)**

**1. Group by one column: Region**

**grouped\_region = df.groupby('Region')['Sales'].sum()**

**print("1️⃣ Total Sales by Region:\n", grouped\_region)**

**Region**

**East 800**

**South 500**

**West 500**

**Name: Sales, dtype: int64**

**. Group by multiple columns: Region + Salesperson**

**grouped\_both = df.groupby(['Region', 'Salesperson'])['Sales'].sum()**

**print("2️⃣ Sales by Region and Salesperson:\n", grouped\_both)**

**Region Salesperson**

**East Alice 400**

**Charlie 400**

**South David 500**

**West Bob 500**

**Name: Sales, dtype: int64**

**3. Use .agg() to apply multiple functions**

**summary = df.groupby('Region')['Sales'].agg(['sum', 'mean', 'count'])**

**print("3️⃣ Summary per Region:\n", summary)**

**sum mean count**

**Region**

**East 800 266.67 3**

**South 500 500.00 1**

**West 500 250.00 2**

| **Task** | **Code Example** |
| --- | --- |
| **Group by column** | **df.groupby('Col')** |
| **Group by and sum** | **df.groupby('Col')['Val'].sum()** |
| **Group by multiple columns** | **df.groupby(['Col1', 'Col2'])['Val'].sum()** |
| **Multiple agg functions** | **df.groupby('Col')['Val'].agg(['sum', 'mean'])** |
| **Reset index** | **df.groupby(...).sum().reset\_index()** |

**Topic 7: Concat and Merge in Pandas**

**Why Use concat() and merge()?**

* **concat(): Stacks DataFrames vertically (rows) or horizontally (columns).**
* **merge(): Joins two DataFrames based on common columns (like SQL joins).**

**✅ 1. concat() – Combine Along Axis**

**import pandas as pd**

**# Create two DataFrames with same columns**

**df1 = pd.DataFrame({**

**'Name': ['Alice', 'Bob'],**

**'Age': [25, 30]**

**})**

**df2 = pd.DataFrame({**

**'Name': ['Charlie', 'David'],**

**'Age': [22, 28]**

**})**

**# Vertical concat (like stacking rows)**

**df\_concat = pd.concat([df1, df2], ignore\_index=True)**

**print("1️⃣ Vertical concat:\n", df\_concat)**

**Name Age**

**0 Alice 25**

**1 Bob 30**

**2 Charlie 22**

**3 David 28**

**df3 = pd.DataFrame({**

**'Score': [85, 90, 88, 92]**

**})**

**✅ Horizontal concat (side-by-side columns)**

**df\_hconcat = pd.concat([df\_concat, df3], axis=1)**

**print("2️⃣ Horizontal concat:\n", df\_hconcat)**

**Name Age Score**

**0 Alice 25 85**

**1 Bob 30 90**

**2 Charlie 22 88**

**3 David 28 92**

**✅ 2. merge() – SQL-style JOINs**

**students = pd.DataFrame({**

**'ID': [1, 2, 3],**

**'Name': ['Alice', 'Bob', 'Charlie']**

**})**

**marks = pd.DataFrame({**

**'ID': [1, 2, 4],**

**'Score': [85, 90, 95]**

**})**

**df\_merge = pd.merge(students, marks, on='ID', how='inner') # INNER JOIN**

**print("3️⃣ Inner Join:\n", df\_merge)**

**ID Name Score**

**0 1 Alice 85**

**1 2 Bob 90**

**Other Merge Types:**

**# LEFT JOIN**

**df\_left = pd.merge(students, marks, on='ID', how='left')**

**print("4️⃣ Left Join:\n", df\_left)**

**# RIGHT JOIN**

**df\_right = pd.merge(students, marks, on='ID', how='right')**

**print("5️⃣ Right Join:\n", df\_right)**

**# OUTER JOIN**

**df\_outer = pd.merge(students, marks, on='ID', how='outer')**

**print("6️⃣ Outer Join:\n", df\_outer)**

| **Task** | **Method** | **Use case** |
| --- | --- | --- |
| **Stack rows** | **pd.concat([...])** | **Combine same-structure data** |
| **Stack columns** | **pd.concat([...], axis=1)** | **Add columns side-by-side** |
| **SQL-style join** | **pd.merge(df1, df2)** | **Combine with keys** |
| **Inner Join** | **merge(..., how='inner')** | **Keep matching rows** |
| **Left Join** | **merge(..., how='left')** | **All left + match from right** |
| **Outer Join** | **merge(..., how='outer')** | **All data, fill missing with NaN** |

**Topic 8: Data Visualization Using Matplotlib and Seaborn**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**data = {**

**'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],**

**'Age': [25, 30, 22, 28, 26],**

**'Score': [85, 90, 78, 88, 82],**

**'Gender': ['F', 'M', 'M', 'M', 'F']**

**}**

**df = pd.DataFrame(data)**

**plt.plot(df['Name'], df['Score'], marker='o')**

**plt.title("Score by Student")**

**plt.xlabel("Name")**

**plt.ylabel("Score")**

**plt.grid(True)**

**plt.show()**

**✅ 1. Line Plot (using Matplotlib)**

**python**

**CopyEdit**

**plt.plot(df['Name'], df['Score'], marker='o')**

**plt.title("Score by Student")**

**plt.xlabel("Name")**

**plt.ylabel("Score")**

**plt.grid(True)**

**plt.show()**

**2. Bar Plot (using Seaborn)**

**python**

**CopyEdit**

**sns.barplot(x='Name', y='Score', data=df)**

**plt.title("Bar Plot of Scores")**

**plt.show()**

**✅ 3. Histogram – Distribution of values**

**plt.hist(df['Score'], bins=5, color='skyblue')**

**plt.title("Score Distribution")**

**plt.xlabel("Score")**

**plt.ylabel("Frequency")**

**plt.show()**

**✅ 4. Box Plot – Summary statistics + outliers**

**python**

**CopyEdit**

**sns.boxplot(x='Gender', y='Score', data=df)**

**plt.title("Score Distribution by Gender")**

**plt.show()**

**5. Scatter Plot – Relationship between 2 numerical features**

**python**

**CopyEdit**

**sns.scatterplot(x='Age', y='Score', data=df, hue='Gender')**

**plt.title("Age vs Score")**

**plt.show()**

**6. Correlation Heatmap (using Seaborn)**

**python**

**CopyEdit**

**sns.heatmap(df.corr(), annot=True, cmap='coolwarm')**

**plt.title("Correlation Matrix")**

**plt.show()**

| **Chart Type** | **Use Case** | **Library** |
| --- | --- | --- |
| **Line Plot** | **Trends over categories/time** | **matplotlib** |
| **Bar Plot** | **Compare categories** | **seaborn** |
| **Histogram** | **Distribution of one variable** | **matplotlib** |
| **Box Plot** | **Summary + outliers** | **seaborn** |
| **Scatter Plot** | **Relationship between two variables** | **seaborn** |
| **Heatmap** | **Correlation between numeric columns** | **seaborn** |