🔶 What is `groupby` in Pandas?

`groupby()` is used to split the data into groups based on some criteria, then apply a function (like aggregation, transformation, or filtering), and finally combine the results.

Syntax:

```
df.groupby(by='column_name')
```

🧪 Dataset for Demonstration

```
import pandas as pd

data = {
    'Department': ['Sales', 'Sales', 'HR', 'HR', 'IT', 'IT', 'Sales', 'IT',
'HR'],
    'Employee': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'],
    'Salary': [50000, 52000, 58000, 60000, 65000, 70000, 51000, 69000, 62000],
    'Experience': [2, 3, 5, 6, 7, 8, 3, 7, 5],
    'Rating': [4, 5, 4, 3, 4, 5, 3, 5, 4]
}

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

📊 Basic `groupby` with Aggregation (`agg`)

➤ Group by single column

```
grouped = df.groupby('Department')
grouped['Salary'].mean()
```

➤ Using `.agg()` with multiple functions

```
grouped.agg({
    'Salary': ['mean', 'sum', 'max', 'min'],
    'Experience': ['mean', 'std'],
    'Rating': 'count'
})
```

➕ All Math Functions You Can Use in `agg()`

Function	Description
`mean()`	Average
`sum()`	Sum of values
`min()` / `max()`	Minimum / Maximum
`std()`	Standard deviation
`var()`	Variance
`count()`	Non-null count
`median()`	Median value
`nunique()`	Count of unique values

🔧 `groupby` Attributes & Methods

1. 🔢 Total number of groups

```
len(grouped)
```

2. 📦 Items in each group

```
grouped.size()
```

3. 📌 Get first/last item of each group

```
grouped.first()
grouped.last()
```

4. 🔍 Get a specific group

```
grouped.get_group('HR')
```

5. 📑 Access underlying group mapping

```
grouped.groups
```

6. 📊 Describe each group

```
grouped.describe()
```

7. 🎲 Sample from each group

```
grouped.sample(1)
```

8. 🔢 Number of unique values per group

```
grouped.nunique()
```

9. 🎯 N-th item from each group

```
grouped.nth(1)
```

🔄 `apply()` on groupby

You can apply a custom function to each group:

```
grouped.apply(lambda x: x['Salary'].mean())
```

🔄 `reset_index()`

After a `groupby`, to flatten the multi-index:

```
grouped.agg('mean').reset_index()
```

🧠 `replace=True` in `groupby.nth()`

If `dropna=False` and `replace=True`, it replaces missing values with the nearest available values. Rarely used. Example:

```
grouped.nth(2, dropna='any')
```

✅ Example Questions & Solutions

Q1. Get department-wise average salary and rating.

```
df.groupby('Department').agg({
    'Salary': 'mean',
    'Rating': 'mean'
})
```

Q2. Which department has the highest average experience?

```
df.groupby('Department')['Experience'].mean().sort_values(ascending=False)
```

Q3. Count of employees per department

```
df.groupby('Department')['Employee'].count()
```

Q4. Get the second most experienced employee from each department

```
df.groupby('Department').apply(lambda x: x.sort_values('Experience',
ascending=False).iloc[1])
```

Q5. Describe the 'IT' department employees

```
df.groupby('Department').get_group('IT').describe()
```

Q6. Find departments with total salary > 160000

```
df.groupby('Department')['Salary'].sum().loc[lambda x: x > 160000]
```

Q7. Select all employees with the highest rating in each department

```
df[df.groupby('Department')['Rating'].transform('max') == df['Rating']]
```

💡 Summary Table

Method	Use Case
`groupby().mean()`	Compute average values per group
`.agg()`	Apply multiple aggregations
`.get_group()`	Access a specific group
`.size()`	Number of items per group
`.first()`	First row in each group
`.nth(n)`	N-th row in each group

Method	Use Case
`.apply()`	Apply custom function to each group
`.reset_index()`	Flatten MultiIndex after groupby
`.sample()`	Random sample per group

🧩 Pandas Data Merging, Joining, and Concatenating – Complete Notes

---

🔷 Introduction

Pandas provides powerful tools to **combine** multiple DataFrames using:

1. `concat()` – Concatenate along a particular axis
2. `merge()` – SQL-style joins (inner, left, right, outer)
3. `join()` – Convenient method to join on indexes or keys

🧮 1. Concatenation using `pd.concat()`

📖 Theory:

- Used to **stack** DataFrames vertically (rows) or horizontally (columns).
- **No matching keys required**.
- Useful when combining datasets with same columns or same indexes.

🔣 Syntax:

```
pd.concat(objs, axis=0, join='outer', ignore_index=False)
```

Parameter	Description
`objs`	List/tuple of DataFrames
`axis`	`0` = row-wise (default), `1` = column-wise
`join`	`inner` or `outer` join on axes

Parameter	Description
`ignore_index`	If `True`, reindex the result

🧪 Example 1: Vertical Concatenation (row-wise)

```
import pandas as pd

df1 = pd.DataFrame({'ID': [1, 2], 'Name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'ID': [3, 4], 'Name': ['Charlie', 'David']})

pd.concat([df1, df2], ignore_index=True)
```

✅ Output:

ID	Name
1	Alice
2	Bob
3	Charlie
4	David

🧪 Example 2: Horizontal Concatenation (column-wise)

```
df3 = pd.DataFrame({'Marks': [85, 90]})
pd.concat([df1, df3], axis=1)
```

✅ Output:

ID	Name	Marks
1	Alice	85
2	Bob	90

🧪 Example 3: Inner Join with `concat`

```
df4 = pd.DataFrame({'ID': [1, 2], 'Age': [23, 25]}, index=[0, 1])
df5 = pd.DataFrame({'Gender': ['F', 'M']}, index=[1, 2])

pd.concat([df4, df5], axis=1, join='inner')
```

✅ Output includes only the common index:

ID	Age	Gender
1	25	M

🔷 2. Merging using `pd.merge()`

📖 Theory:

- Used to **combine two DataFrames based on common columns or keys**.
- SQL-style join operations: `inner`, `left`, `right`, `outer`.
- Default join type is `inner`.

🔣 Syntax:

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None)
```

Parameter	Description
`left`, `right`	DataFrames to merge
`how`	Type of join: `'left'`, `'right'`, `'outer'`, `'inner'`
`on`	Column name(s) to join on

👥 Types of Joins Explained with Illustration

Let's define:

```
left = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})

right = pd.DataFrame({
    'ID': [2, 3, 4],
    'Score': [90, 85, 88]
})
```

➤ Inner Join (only common IDs)

```
pd.merge(left, right, on='ID', how='inner')
```

ID	Name	Score
2	Bob	90
3	Charlie	85

➤ Left Join (all from left, fill missing from right)

```
pd.merge(left, right, on='ID', how='left')
```

ID	Name	Score
1	Alice	NaN
2	Bob	90
3	Charlie	85

➤ Right Join (all from right, fill missing from left)

```
pd.merge(left, right, on='ID', how='right')
```

ID	Name	Score
2	Bob	90
3	Charlie	85
4	NaN	88

➤ Outer Join (all from both, fill NaNs)

```
pd.merge(left, right, on='ID', how='outer')
```

ID	Name	Score
1	Alice	NaN
2	Bob	90
3	Charlie	85
4	NaN	88

🧪 Merging on different column names

```
df1 = pd.DataFrame({'emp_id': [1, 2], 'name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'id': [1, 2], 'dept': ['HR', 'IT']})

pd.merge(df1, df2, left_on='emp_id', right_on='id')
```

🔷 3. `join()` Method

📖 Theory:

- A shortcut for merging on **index**.
- Useful when one of the DataFrames has an index you want to join on.

🔣 Syntax:

```
df1.join(df2, how='left')
```

🧪 Example

```
df1 = pd.DataFrame({'Name': ['Alice', 'Bob']}, index=[1, 2])
df2 = pd.DataFrame({'Score': [85, 90]}, index=[1, 2])

df1.join(df2)
```

✅ Output:

	Name	Score
1	Alice	85
2	Bob	90

🧠 Summary Table

Operation	Function	Joins on	Default Join	Best for...
Concatenation	`pd.concat()`	Axis/index	Outer	Stacking DataFrames
Merge	`pd.merge()`	Columns/key	Inner	SQL-style complex joins
Join	`df.join()`	Index	Left	Simple joins on index

📝 Practice Problems

Q1. Combine student details and their marks into one DataFrame.

```
students = pd.DataFrame({'ID': [1,2,3], 'Name': ['A', 'B', 'C']})
marks = pd.DataFrame({'ID': [2,3,4], 'Maths': [85, 90, 80]})
```

```
pd.merge(students, marks, on='ID', how='outer')
```

Q2. Concatenate monthly sales data into one big DataFrame.

```
jan = pd.DataFrame({'Day': [1,2], 'Sales': [100, 200]})
feb = pd.DataFrame({'Day': [1,2], 'Sales': [150, 180]})

pd.concat([jan, feb], keys=['Jan', 'Feb'])
```

Q3. Join department names with employee data using indexes.

```
emp = pd.DataFrame({'Name': ['Alice', 'Bob']}, index=[101, 102])
dept = pd.DataFrame({'Dept': ['HR', 'IT']}, index=[101, 102])

emp.join(dept)
```

🧩 Attributes/Parameters of `concat()`, `merge()`, and `join()` in Pandas

🔶 1. `pd.concat()` – Attributes Explained

🔣 Syntax:

```
pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None,
names=None, verify_integrity=False, sort=False)
```

📋 Parameters:

Parameter	Description
`objs`	List or tuple of DataFrames or Series to concatenate
`axis`	`0` for row-wise (default), `1` for column-wise
`join`	How to handle indexes: `'outer'` (default), `'inner'` (intersection)
`ignore_index`	If `True`, index will be reset in the result
`keys`	Used to create hierarchical index (MultiIndex) for identifying original pieces
`names`	Names for the levels in the new hierarchical index
`verify_integrity`	If `True`, checks for duplicates and raises error
`sort`	If `True`, sorts the result columns/rows (helps with mixed indexes)

🧪 Example with keys and MultiIndex:

```
df1 = pd.DataFrame({'A': [1, 2]})
df2 = pd.DataFrame({'A': [3, 4]})

pd.concat([df1, df2], keys=['Group1', 'Group2'])
```

✅ Output:

	A
Group1	
0	1
1	2
Group2	
0	3
1	4

🔷 2. `pd.merge()` – Attributes Explained

🔣 Syntax:

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
         left_index=False, right_index=False, sort=False, suffixes=('_x',
'_y'), indicator=False, validate=None)
```

📋 Parameters:

Parameter	Description
`left`, `right`	DataFrames to merge
`how`	Type of join: `'left'`, `'right'`, `'outer'`, `'inner'` (default: inner)
`on`	Column(s) to join on (must exist in both DataFrames)
`left_on` / `right_on`	Column(s) from left/right DataFrame to use as join key
`left_index` / `right_index`	Use index as join key (boolean)
`sort`	Sort the result DataFrame by join key
`suffixes`	Tuple for overlapping column names in left/right
`indicator`	Adds a column `_merge` indicating source of each row
`validate`	Used to check for valid merge conditions (like one-to-one, many-to-one, etc.)

🧪 Example: Using `indicator` and `suffixes`

```
left = pd.DataFrame({'ID': [1, 2], 'Name': ['A', 'B']})
right = pd.DataFrame({'ID': [2, 3], 'Name': ['X', 'Y']})

pd.merge(left, right, on='ID', how='outer', suffixes=('_left', '_right'),
indicator=True)
```

✅ Output:

ID	Name_left	Name_right	_merge
1	A	NaN	left_only
2	B	X	both
3	NaN	Y	right_only

`validate` usage:

Ensures your join condition matches expectations.

```
pd.merge(left, right, on='ID', validate='one_to_one')  # Error if duplicates exist
```

🟢 3. `df.join()` – Attributes Explained

🔣 Syntax:

```
df1.join(df2, on=None, how='left', lsuffix='', rsuffix='', sort=False)
```

📋 Parameters:

Parameter	Description
`on`	Column name from left DataFrame to join on (used if joining on column, not index)
`how`	Type of join: `'left'` (default), `'right'`, `'outer'`, `'inner'`
`lsuffix`, `rsuffix`	Suffixes to apply to overlapping column names
`sort`	Sort the result DataFrame by join key

🧪 Example with index-based join

```
df1 = pd.DataFrame({'Name': ['Alice', 'Bob']}, index=[1, 2])
df2 = pd.DataFrame({'Score': [90, 85]}, index=[1, 2])

df1.join(df2)
```

✅ Output:

	Name	Score
1	Alice	90
2	Bob	85

🧪 Example with `on` (column join)

```
df1 = pd.DataFrame({'ID': [1, 2], 'Name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'ID': [1, 2], 'Score': [90, 95]})

df1.set_index('ID').join(df2.set_index('ID'))
```

🧠 1. What is a MultiIndex?

A **MultiIndex** allows multiple levels of indexing (hierarchical indexing) on Series or DataFrames. This is useful for working with complex, multi-dimensional data.

📘 2. MultiIndex in Series

🔹 Creating a MultiIndex Series

```
import pandas as pd

index = pd.MultiIndex.from_tuples(
    [('A', 2020), ('A', 2021), ('B', 2020), ('B', 2021)],
    names=['Company', 'Year']
)
data = pd.Series([100, 120, 90, 110], index=index)
print(data)
```

🔹 Accessing Elements

```
data['A']
data.loc[('A', 2020)]
```

📗 3. MultiIndex in DataFrame

◆ Creating a MultiIndex DataFrame

```
arrays = [
    ['Math', 'Math', 'English', 'English'],
    ['Test1', 'Test2', 'Test1', 'Test2']
]
index = pd.MultiIndex.from_arrays(arrays, names=['Subject', 'Test'])

df = pd.DataFrame([[85, 90, 78, 80], [88, 92, 74, 82]],
                  index=['Alice', 'Bob'],
                  columns=index)

print(df)
```

◆ Accessing Columns and Rows

```
# Access all English scores
df['English']

# Access Bob's Test2 scores in Math
df.loc['Bob', ('Math', 'Test2')]
```

🔄 4. Stacking & Unstacking

◆ `stack()` – Columns → Index

```
stacked = df.stack()  # Stack last level by default
print(stacked)
```

◆ `unstack()` – Index → Columns

```
unstacked = stacked.unstack()
```

🔹 Specifying Level

```
df.stack(level=0)  # Stack first column level
```

🧪 5. Working with MultiIndexed DataFrames

🔹 Accessing by Level Name

```
df.loc[:, ('Math', slice(None))]   # All Math tests
df.loc[:, pd.IndexSlice['Math', :]]
```

🔹 Swapping and Sorting Index Levels

```
# Swap levels
df.columns = df.columns.swaplevel()

# Sort index
df = df.sort_index(axis=1)
```

🔹 Setting and Resetting Index

```
df_reset = df.reset_index()
df_new = df_reset.set_index(['Subject', 'Test'])
```

🔁 6. Transpose with MultiIndex

```
df_T = df.T  # Transpose rows ⇔ columns
print(df_T.index.names)  # ['Subject', 'Test']
```

Useful when comparing student-wise vs test-wise results.

---

7. Long vs Wide Format

Wide Format

Each variable gets a column. (Default DataFrame style)

Long Format with `melt()`

```
df_long = df_reset.melt(id_vars='index', var_name=['Subject', 'Test'],
value_name='Score')
```

Or if from a non-multiindex DF:

```
df_long = pd.melt(df, id_vars=['Student'], var_name='Subject_Test',
value_name='Score')
```

---

8. Pivot & Pivot Table

`pivot()`

Rearranges data without aggregation.

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Alice', 'Bob'],
    'Subject': ['Math', 'Math', 'English', 'English'],
    'Score': [90, 85, 95, 80]
})
df.pivot(index='Name', columns='Subject', values='Score')
```

`pivot_table()`

Supports aggregation like mean, sum, etc.

```
df.pivot_table(index='Name', columns='Subject', values='Score',
aggfunc='mean')
```

Use `margins=True` to show totals.

💡 9. Other Tips You Should Know

🔸 `xs()` – Cross Section (slicing one level)

```
data.xs(key='A', level='Company')  # Get all years for A
df.xs('Math', axis=1, level=0)     # Get all Math tests
```

🔸 `groupby()` with MultiIndex

```
df_reset.groupby(['Subject', 'Test'])['Score'].mean()
```

🔸 Rename Levels

```
df.columns.names = ['Subject', 'Test']
df.index.names = ['Student']
```

📚 Summary Table

Feature	Method/Function	Use Case
MultiIndex	`pd.MultiIndex.from_*`	Create hierarchical index
Stack	`.stack(level=)`	Collapse column → index
Unstack	`.unstack(level=)`	Expand index → column
Cross Section	`.xs()`	Slice using level value
Long Format	`pd.melt()`	Normalize data into long form
Wide Format	`pivot()`	Spread data across columns
Aggregated Pivot	`pivot_table()`	Same as pivot + aggregation
Index tools	`swaplevel`, `sort_index`, `set_index`, `reset_index`	Manipulate multi-index