Pandas – Quick Revision Guide for Beginners

Overview

Pandas is a Python library for working with **tables** of data (like Excel). You can load, look at, filter, sort, and clean data. This guide goes from **basic to advanced** in a logical order.

Format used for each topic:

- **Description** – Short, simple explanation in easy words.
- **Syntax** – The correct Pandas code pattern.
- **Example** – DataFrame **before** → **code** → DataFrame **after** (result).

Table of Contents (Basic → Advanced)

1. Installation and Import

Description: You need to install pandas on your computer and then import it in your code. We use the short name `pd` so we don't have to type "pandas" every time.

Syntax:

```
pip install pandas
```

```
import pandas as pd
```

Example: Run `pip install pandas` in the terminal once. Then in your script write `import pandas as pd`. After that you can use `pd.DataFrame()`, `pd.read_csv()`, etc.

2. Series – One Column of Data

Description: A **Series** is one column of data: a list of values with labels (index). Think of it as a single column from a table.

Syntax:

```
pd.Series(data, index=optional_index)
```

Example – create a Series from a list:

Before: We have a list of ages and want a Series.

Code:

```
import pandas as pd
ages = pd.Series([25, 30, 35, 28])
print(ages)
```

After (result):

```
0    25
1    30
2    35
3    28
dtype: int64
```

Example 1 – from a list:

```
import pandas as pd

ages = pd.Series([25, 30, 35, 28])
print(ages)
# 0    25
# 1    30
# 2    35
# 3    28
```

What it does: Creates a Series with default index 0, 1, 2, 3. Each value has a label (the index).

Example 2 – custom index:

```
temps = pd.Series([22, 24, 19], index=['Mon', 'Tue', 'Wed'])
print(temps)
# Mon    22
# Tue    24
# Wed    19
```

What it does: Row labels are now 'Mon', 'Tue', 'Wed'. You can access by index: `temps['Tue']` gives 24.

Example 3 – access by position or label:

```
print(temps[0])      # 22  (first value)
print(temps['Mon'])  # 22  (by label)
```

3. DataFrame – Tables of Data

Description: A **DataFrame** is a table with rows and columns (like Excel). Each column is a Series. Rows = one record each; columns = different pieces of info (name, age, etc.).

No syntax here – it's just the main object you work with in pandas.

4. Creating DataFrames

Description: You can build a DataFrame from a dictionary: each key is a column name, each value is a list of values for that column. All lists must have the same length.

Syntax:

```
pd.DataFrame({'col1': [val1, val2, ...], 'col2': [val1, val2, ...]})
```

Example – from a dictionary of lists:

Before: We want a table with Name, Age, and City.

Code:

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['NYC', 'LA', 'Chicago']
})
print(df)
```

After (result):

	Name	Age	City
0	Alice	25	NYC
1	Bob	30	LA
2	Charlie	35	Chicago

Example 1 – from dict of lists:

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['NYC', 'LA', 'Chicago']
})
print(df)
#       Name  Age     City
# 0    Alice   25     NYC
# 1      Bob   30      LA
# 2  Charlie   35  Chicago
```

What it does: Keys = column names, values = lists (one per column). All lists must have the same length.

Example 2 – from list of dicts:

```
df = pd.DataFrame([
    {'Name': 'Alice', 'Age': 25},
    {'Name': 'Bob', 'Age': 30},
    {'Name': 'Charlie', 'Age': 35}
])
print(df)
```

What it does: Each dictionary is one row. Missing keys become NaN in that column.

Example 3 – with custom index:

```
df = pd.DataFrame(
    {'A': [1, 2, 3], 'B': [4, 5, 6]},
    index=['x', 'y', 'z']
)
print(df)
#    A  B
# x  1  4
# y  2  5
# z  3  6
```

Example 4 – from list of lists with column names:

You can pass a **list of lists** (each inner list = one row) and set column names with the **columns** parameter.

Syntax:

```
pd.DataFrame(
    [
        [val1, val2, val3],   # row 1
        [val1, val2, val3],   # row 2
        ...
    ],
    columns=['Column1', 'Column2', 'Column3']
)
```

Command:

```
df2 = pd.DataFrame([
    [1, 'San Diego', 100],
    [2, 'Los Angeles', 120],
    [3, 'San Francisco', 90],
    [4, 'Sacramento', 115],
],
    columns=[
        'Store ID',
        'Location',
        'Number of Employees'
    ]
)
print(df2)
#    Store ID     Location  Number of Employees
# 0         1    San Diego                  100
# 1         2  Los Angeles                  120
# 2         3 San Francisco                   90
# 3         4   Sacramento                  115
```

What it does: Each inner list is one **row** of data. The **columns** argument gives the column names in order. The number of column names must match the number of values per row. Handy when you have row-wise data (e.g. from a file or a table) and want to assign names in one place.

5. Reading and Writing Files

Description: You can load a table from a CSV or Excel file into a DataFrame. You can also save a DataFrame back to a file.

Syntax:

```
pd.read_csv('file_path.csv')
pd.read_excel('file_path.xlsx', sheet_name='Sheet1')
df.to_csv('output.csv', index=False)
df.to_excel('output.xlsx', index=False)
```

Example – read CSV:

Before: A file `employees.csv` exists on your computer.

Code:

```
df = pd.read_csv('employees.csv')
```

After: `df` is a DataFrame with the data from the file. First row of the file becomes column names by default.

Example – save to CSV:

Before: You have a DataFrame `df`.

Code:

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

After: A file `output.csv` is created. `index=False` means row numbers are not written as a column.

Common read options:

```
df = pd.read_csv('data.csv', sep=';')           # separator is ; not ,
df = pd.read_csv('data.csv', header=0)          # row 0 = column names
df = pd.read_csv('data.csv', index_col=0)       # first column = index
df = pd.read_csv('data.csv', usecols=['A','B']) # only these columns
```

6. Viewing Data (head, tail, info, describe)

Description: These methods let you peek at your data: first or last rows, column types, missing values, and basic numbers (mean, min, max). Use them to understand the table before doing more.

Syntax:

```
df.head(n)      # first n rows (default 5)
df.tail(n)      # last n rows (default 5)
df.info()       # column names, types, non-null counts
df.describe()   # mean, min, max, etc. for number columns
df.shape        # (rows, columns)
df.columns      # list of column names
```

Example – head() and the guide's sample DataFrame:

Before: We have a DataFrame. We want to see the first few rows.

Code:

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve'],
    'Age': [25, 30, 35, 28, 32],
    'Salary': [50000, 60000, 55000, 52000, 58000],
    'Dept': ['HR', 'IT', 'IT', 'HR', 'IT']
})
print(df.head())
```

After (result):

	Name	Age	Salary	Dept
0	Alice	25	50000	HR
1	Bob	30	60000	IT
2	Charlie	35	55000	IT
3	Diana	28	52000	HR
4	Eve	32	58000	IT

Quick reference: `df.tail(3)` = last 3 rows. `df.info()` = types and null counts. `df.describe()` = mean, min, max for numeric columns. `df.shape` = (rows, cols). `df.columns` = column names. `df.dtypes` = type of each column.

7. Selecting Columns and Rows

Description: You can pick one or more columns, or one or more rows. One column gives a Series; multiple columns or rows give a DataFrame. Use **iloc** for position (0, 1, 2…) and **loc** for labels.

Syntax:

```
df['col']           # one column (Series)
df[['col1','col2']] # multiple columns (DataFrame)
df.iloc[0]          # 1st row by position
df.iloc[1:4]        # rows 1, 2, 3 (end 4 not included)
df.loc[0]           # row with index 0 (by label)
```

Example – select one column and first 3 rows:

Before: DataFrame `df` with Name, Age, Salary, Dept (5 rows).

Code:

```
df['Name']        # one column
df[['Name','Age']]  # two columns
df.iloc[:3]       # first 3 rows
```

After: `df['Name']` gives a Series (Alice, Bob, Charlie, Diana, Eve). `df.iloc[:3]` gives a DataFrame with rows 0, 1, 2 (Alice, Bob, Charlie).

Selecting rows (details)

DataFrames are zero-indexed: Rows are counted starting from **0**. So the 1st row is index 0, the 2nd row is index 1, the 3rd row is index 2, and so on.

Select a single row by position (iloc):

To get one row by its **position** (0, 1, 2, …), use **iloc**.

```
df.iloc[0]   # 1st row (Alice)
df.iloc[2]   # 3rd row (Charlie – position 2)
```

Example: With our **df** (Name, Age, Salary, Dept), the 1st row is Alice, the 2nd is Bob, and the **3rd row** is Charlie (position 2). To select Charlie's row:

```
df.iloc[2]
# Name       Charlie
# Age            35
# Salary      55000
# Dept           IT
# Name: 2, dtype: object
```

What it does: Returns that one row. When you select a **single row**, the result is a **Series** (just like when you select a single column).

What does "index = column names, values = that row's values" mean?
When you take **one row** from a table, you get a single list of cells: one value per column. In a Series, the **labels on the left** (the "index") are the **column names** (Name, Age, Salary, Dept), and the **numbers or text on the right** (the "values") are **that row's data**. So for Charlie's row you see:

- **Index (labels):** Name, Age, Salary, Dept
- **Values:** Charlie, 35, 55000, IT

Think of it as: the column names become the **labels** of the Series, and the row's cells become the **values** of the Series.

Selecting multiple rows

You can select **more than one row** at a time using **iloc** and a **slice** (start : end). With slices, the **end index is not included** (same as Python lists). Our **df** has 5 rows: index 0 = Alice, 1 = Bob, 2 = Charlie, 3 = Diana, 4 = Eve.

1. Slice `iloc[start:end]` – rows from position start up to but not including end

```
df.iloc[1:4]
```

What it does: Selects rows at positions **1, 2, and 3** (Bob, Charlie, Diana). Position 4 (Eve) is **not** included. So: start at the 2nd row, up to but not including the 5th row.

Name	Age	Salary	Dept
Bob	30	60000	IT
Charlie	35	55000	IT
Diana	28	52000	HR

2. Slice `iloc[:n]` – first n rows (from the start up to but not including position n)

```
df.iloc[:3]
```

What it does: Selects all rows **up to but not including** position 3. So you get positions **0, 1, 2** (Alice, Bob, Charlie) – the first 3 rows.

3. Slice `iloc[-n:]` – last n rows

```
df.iloc[-2:]
```

What it does: -2 means "2nd from the end," and **:** means "to the end." So you get the **last 2 rows** (Diana, Eve). In general, **iloc[-n:]** gives you the last n rows of the DataFrame.

Quick recap:

Code	Meaning	With our df (5 rows)
`df.iloc[1:4]`	Rows 1, 2, 3 (end 4 not included)	Bob, Charlie, Diana
`df.iloc[:3]`	First 3 rows (0, 1, 2)	Alice, Bob, Charlie
`df.iloc[-2:]`	Last 2 rows	Diana, Eve

loc – select rows (and columns) by label

Use **loc** when you want to use **row and column names** (labels). With **loc**, a slice like `1:3` includes **both** 1 and 3 (inclusive).

```
# Row with index 0, all columns
df.loc[0]

# Row 0, only columns 'Name' and 'Age'
df.loc[0, ['Name', 'Age']]

# Rows 0 and 2, only column 'Name'
df.loc[[0, 2], 'Name']

# Rows 1 to 3 (inclusive), columns 'Name' and 'Salary'
df.loc[1:3, ['Name', 'Salary']]
```

What it does: `loc` uses **labels** (index and column names). Slices are **inclusive** on both ends (unlike iloc).

iloc – select rows (and columns) by position

Use **iloc** when you want to use **integer positions** (0, 1, 2, …). Slices are **exclusive** of the end (like Python).

```
# First row (position 0)
df.iloc[0]

# Row at position 0, columns at positions 0 and 2
df.iloc[0, [0, 2]]

# Rows 0 to 2 (position 3 not included), all columns
df.iloc[0:3]

# Rows 1 and 2, columns 1 and 2
df.iloc[1:3, 1:3]
```

What it does: `iloc` uses **integer positions** (0-based). Slices are **exclusive** of the end (e.g. 0:3 gives 0, 1, 2).

Quick comparison:

Need	Use	Example
One column	`df['col']`	`df['Name']`
Multiple columns	`df[[]]`	`df[['Name','Age']]`
Rows by label	`loc`	`df.loc[0]` , `df.loc[1:3, 'Name']`
Rows by position	`iloc`	`df.iloc[0]` , `df.iloc[0:3]`

8. Filtering Rows (boolean indexing)

Description: Filtering means **keeping only rows** that meet a condition (e.g. Age > 28, or Dept = 'IT'). You put the condition inside square brackets: `df[condition]` .

Syntax:

```
df[df['col'] > value]      # greater than
df[df['col'] == value]     # equals
df[(cond1) & (cond2)]      # AND (both true)
df[(cond1) | (cond2)]      # OR (at least one true)
df[df['col'].isin([a,b])]  # value in list
```

Example – keep only IT department:

Before: DataFrame `df` with 5 rows (Alice HR, Bob IT, Charlie IT, Diana HR, Eve IT).

Code:

```
df[df['Dept'] == 'IT']
```

After (result): A DataFrame with 3 rows (Bob, Charlie, Eve).

	Name	Age	Salary	Dept
1	Bob	30	60000	IT
2	Charlie	35	55000	IT
4	Eve	32	58000	IT

More: Use `&` for AND, `|` for OR (put each condition in parentheses). Use `df['col'].isin(['A','B'])` to match any value in a list.

9. Adding and Dropping Columns

Description: You can add new columns (from a list, from a calculation, or with the same value for all rows) or remove columns and rows. You can also rename columns and reset the row index.

Syntax (common operations):

```
df['new_col'] = values_or_expression   # add column
df.drop(columns=['col'], inplace=False) # drop column
df.rename(columns={'old':'new'})       # rename column
df.reset_index(drop=True)               # new index 0,1,2,...
```

Example – add a new column (same value for all rows):

Before: DataFrame `df` with columns Name, Age, Salary, Dept.

Code:

```
df['Employment_Type'] = 'Full-time'
```

After (result): Same table with an extra column; every row has "Full-time" in that column.

Add a new column from a list (same length as the DataFrame):

You can add a column by assigning a **list** of values (one value per row). The list length must match the number of rows.

```
df['Location'] = ['NYC', 'LA', 'Chicago', 'Boston', 'Seattle']
```

Example result:

Name	Age	Salary	Dept	Location
Alice	25	50000	HR	NYC
Bob	30	60000	IT	LA
Charlie	35	55000	IT	Chicago
Diana	28	52000	HR	Boston
Eve	32	58000	IT	Seattle

Add a new column with the same value for all rows:

You can add a column where **every row** gets the **same** value. Assign a single value (number, string, or boolean) to the new column name.

```
df['Employment_Type'] = 'Full-time'
# or
df['Status'] = 'Active'
```

What it does: Creates a new column and fills **all rows** with that value. Pandas automatically repeats the value for every row. Use this when one piece of information applies to the whole table (e.g. all full-time, all records active).

Example: After `df['Employment_Type'] = 'Full-time'` , every row has the same value in that column:

Name	Age	Salary	Dept	Employment_Type
Alice	25	50000	HR	Full-time
Bob	30	60000	IT	Full-time
Charlie	35	55000	IT	Full-time
Diana	28	52000	HR	Full-time

Name	Age	Salary	Dept	Employment_Type
Eve	32	58000	IT	Full-time

Add a new column by performing a calculation on existing columns:

You can add a column by applying a **calculation** (or function) to existing columns. Each row gets the result of that calculation. For example, like computing sales tax from price, you can compute bonus from salary.

```
df['Bonus'] = df['Salary'] * 0.1
# or combine columns: df['FullInfo'] = df['Name'] + ' - ' + df['Dept']
```

What it does: The new column is computed **row by row** from other columns. Here, **Bonus** = 10% of **Salary** for each row. You can use any expression (multiply, add, combine columns, etc.).

Example with our df: After `df['Bonus'] = df['Salary'] * 0.1` , the table has a **Bonus** column:

Name	Age	Salary	Dept	Bonus
Alice	25	50000	HR	5000.0
Bob	30	60000	IT	6000.0
Charlie	35	55000	IT	5500.0
Diana	28	52000	HR	5200.0
Eve	32	58000	IT	5800.0

Reviewing Lambda Functions

A **lambda** is a small function defined in **one line**. You can assign it to a variable and call it like a normal function. The syntax is: **lambda** *input* **:** *expression* (the result of the expression is returned).

Example 1 – numeric: A lambda that multiplies by 2 and adds 3:

```
mylambda = lambda x: (x * 2) + 3
print(mylambda(5))   # (5 * 2) + 3 = 13
# Output: 13
```

Example 2 – string: A lambda that converts a string to lowercase:

```
stringlambda = lambda x: x.lower()
print(stringlambda("Oh Hi Mark!"))
# Output: "oh hi mark!"
```

What it does: Lambda works with **any type** (numbers, strings, etc.). The value passed in is the **input** (e.g. x); the **expression** on the right is computed and returned. You don't need a separate `def` ; the one line is the whole function.

Using lambda with our DataFrame: We often pass a lambda to **apply()** so the same logic runs on every value in a column. For example, a lambda that doubles the value: `lambda x: x * 2` . Used in a column:
`df['DoubleSalary'] = df['Salary'].apply(lambda x: x * 2)` .

***Reviewing Lambda Function: If Statements**

You can make lambdas more complex by using a **conditional (if/else)** in the expression. The syntax is:

```
lambda x: [OUTCOME IF TRUE] if [CONDITIONAL] else [OUTCOME IF FALSE]
```

Example – "big" or "small" based on a number: Same logic as a normal function, but in one line.

Regular function:

```
def myfunction(x):
    if x > 10:
        return "big"
    else:
        return "small"
```

Equivalent lambda:

```
myfunction = lambda x: "big" if x > 10 else "small"
# e.g. myfunction(15) → "big", myfunction(5) → "small"
```

What it does: If `x > 10` , return `"big"` ; otherwise return `"small"` . The lambda form is just the if/else squeezed into one expression.

Using if/else lambda with our DataFrame: We can use the same pattern on a column. For example, add a column **SalaryTier**: "High" if Salary > 55000, else "Standard":

```
df['SalaryTier'] = df['Salary'].apply(lambda x: "High" if x > 55000 else "Standard")
```

So for our **df** (Alice 50k, Bob 60k, Charlie 55k, …), rows with salary > 55000 get "High", others get "Standard". Same syntax: **value_if_true if condition else value_if_false** inside the lambda.

Add a new column (conditional logic – apply + lambda):

```
df['status'] = df['Age'].apply(lambda x: 'Senior' if x >= 30 else 'Junior')
```

What it does: `apply()` runs a function on each value in the column. Here we use a **lambda**: if age ≥ 30 then 'Senior', else 'Junior'. Use this when the new column depends on row-wise logic (if/else, custom function).

Applying a Lambda to a Column

We often use **lambda** with **apply()** to do more complex operations on a column (e.g. use string methods like **.split()** to extract part of each value).

Example – email provider from email address: Suppose we have a table with **Name** and **Email**. We want a new column with the **email provider** (the part after `@`). Use **.split('@')** and take the last part **[-1]**:

```
df['Email Provider'] = df['Email'].apply(lambda x: x.split('@')[-1])
```

What it does: For each email, `x.split('@')` gives e.g. `['john.smith', 'gmail.com']`; **[-1]** is `'gmail.com'`. So we get the domain. Example result:

Name	Email	Email Provider
JOHN SMITH	john.smith@gmail.com	gmail.com
Jane Doe	jdoe@yahoo.com	yahoo.com
joe schmo	joeschmo@hotmail.com	hotmail.com

With our DataFrame: We don't have an Email column, but we can use the same idea on **Name**. For example, get the **first name** (first word) from each full name:

```
df['FirstName'] = df['Name'].apply(lambda x: x.split()[0])
```

For our **df** (Alice, Bob, Charlie, Diana, Eve), this gives a new column **FirstName** with values Alice, Bob, Charlie, Diana, Eve. Here **.split()** splits on spaces and **[0]** takes the first word. Same pattern: **apply** a **lambda** that uses a string method to build a new column.

Applying a Lambda to a Row

We can operate on **multiple columns at once** by applying a lambda to each **row** instead of each column. Use **apply()** on the whole DataFrame (not on a single column) and set **axis=1**. Then the input to the lambda is an **entire row**. Access values with **row['column_name']** or **row.column_name**.

Example – price with tax (grocery list): Suppose we have **Item**, **Price**, and **Is taxed?**. We want **Price with Tax**: if **Is taxed?** is **Yes**, multiply Price by 1.075 (7.5% tax); otherwise use Price as is.

```
df['Price with Tax'] = df.apply(
    lambda row: row['Price'] * 1.075 if row['Is taxed?'] == 'Yes' else row['Price'],
    axis=1
)
```

What it does: axis=1 means "apply across columns" — i.e. one row at a time. So **row** is one full row; **row['Price']** and **row['Is taxed?']** are that row's values. Each row gets one result for the new column.

With our DataFrame: We can use the same pattern when the new column depends on **two or more columns**. For example, **AdjustedSalary**: give IT a 10% bump, others keep Salary as is:

```
df['AdjustedSalary'] = df.apply(
    lambda row: row['Salary'] * 1.1 if row['Dept'] == 'IT' else row['Salary'],
    axis=1
)
```

For our **df**, IT (Bob, Charlie, Eve) get Salary × 1.1; HR (Alice, Diana) get Salary unchanged. Use **apply(..., axis=1)** whenever your lambda needs **row['Col1']**, **row['Col2']**, etc. in one expression.

Performing column operations (apply with a function):

Sometimes you want to **transform** every value in a column using a function (e.g. change capitalization, round numbers). You can use **apply** with a **built-in or custom function** and assign the result back to the same column to **overwrite** it, or to a new column.

Example – make all names uppercase:

```
df['Name'] = df['Name'].apply(str.upper)
# same as: df['Name'] = df.Name.apply(str.upper)
```

What it does: apply(str.upper) runs the **str.upper** function on every value in the **Name** column. Each name is converted to uppercase. By assigning back to **df['Name']**, we **overwrite** the existing column. Use this when you need a consistent format (e.g. all caps, all lowercase, or a custom function).

With our df – before: Names might be mixed case (Alice, Bob, Charlie, …). **After** `df['Name'] = df['Name'].apply(str.upper)` :

Name	Age	Salary	Dept
ALICE	25	50000	HR
BOB	30	60000	IT
CHARLIE	35	55000	IT
DIANA	28	52000	HR
EVE	32	58000	IT

Other useful functions: `str.lower()` for lowercase, `str.strip()` to remove leading/trailing spaces, or your own function: `df['Col'] = df['Col'].apply(my_function)` .

Drop columns:

```
df.drop(columns=['Bonus'], inplace=False)  # returns new DataFrame
df.drop(['Bonus', 'FullInfo'], axis=1)     # axis=1 means columns
```

Drop rows:

```
df.drop(5, axis=0)       # drop row with index 5
df.drop([0, 2], axis=0)  # drop rows with index 0 and 2
# axis=0 = rows, axis=1 = columns
```

Rename columns:

When data comes from other sources, we often want to change column names — for example so they follow **variable name rules** (no spaces, valid identifiers). Then we can use **df.column_name** (with tab-complete) instead of **df['column_name']**.

Method 1 – .rename (dictionary): Pass a dictionary to the **columns** keyword: `{'old_column_name': 'new_column_name', ...}`. You can rename one or several columns.

```
df = pd.DataFrame({'name': ['John', 'Jane', 'Sue', 'Fred'], 'age': [23, 29, 21, 18]})
df.rename(columns={'name': 'First Name', 'age': 'Age'}, inplace=True)
# Now columns are 'First Name' and 'Age'
```

What it does: Maps old names to new names. Using **rename** without **inplace=True** returns a **new** DataFrame and leaves the original unchanged. **inplace=True** edits the original DataFrame.

With our DataFrame – rename one or two columns:

```
df.rename(columns={'Dept': 'Department'}, inplace=True)   # just one column
df.rename(columns={'Salary': 'sal', 'Dept': 'department'}, inplace=False)  # returns new df
```

Why prefer .rename over .columns? (1) You can rename **just one column**. (2) You are **explicit** about which column gets which name (with **.columns** = list, wrong order can swap labels). (3) If you **misspell** an original column name in the dictionary, **rename** does not raise an error — it simply does not change that column.

Method 2 – set all column names at once: Assign a **list** to **.columns** to replace every column name in one go. Useful when you need to change all names; the **order must match** the current columns, or you will mislabel them.

```
df = pd.DataFrame({'name': ['John', 'Jane', 'Sue', 'Fred'], 'age': [23, 29, 21, 18]})
df.columns = ['First Name', 'Age']   # same order as current columns
```

With our DataFrame: To rename all columns at once (e.g. to lowercase and shorter names), keep the same order as **Name, Age, Salary, Dept**:

```
df.columns = ['name', 'age', 'salary', 'dept']
# Now df.name, df.age, etc. work (valid identifiers)
```

If you wrote `df.columns = ['dept', 'salary', 'age', 'name']` by mistake, the first column (Name) would be labeled "dept" — so double-check the order when using this form.

Reset index (setting indices)

When you select a subset of a DataFrame (e.g. by filtering), the row indices often become **non-consecutive** (e.g. 0, 2, 4 instead of 0, 1, 2). That can look messy and make **.iloc()** harder to use. **reset_index()** gives you a clean index (0, 1, 2, …) again.

Example with our df: Filter so we keep only rows where Dept is 'IT'. The result has indices 1, 2, 4 (Bob, Charlie, Eve), not 0, 1, 2:

```
df_sub = df[df['Dept'] == 'IT']
#   Name     Age  Salary Dept
# 1  Bob      30   60000  IT
# 2  Charlie  35   55000  IT
# 4  Eve      32   58000  IT
```

1. reset_index() (default) – Old indices are moved into a **new column** called `'index'` :

```
df_sub.reset_index()
#   index  Name     Age  Salary Dept
# 0     1  Bob      30   60000  IT
# 1     2  Charlie  35   55000  IT
# 2     4  Eve      32   58000  IT
```

What it does: The new DataFrame has a clean 0, 1, 2 row index, and the **old** indices (1, 2, 4) are stored in a column named `'index'` . You usually don't need that column.

2. reset_index(drop=True) – Clean index and **no** extra column:

```
df_sub.reset_index(drop=True)
#       Name     Age  Salary Dept
# 0      Bob      30   60000  IT
# 1  Charlie      35   55000  IT
# 2      Eve      32   58000  IT
```

What it does: Replaces the index with 0, 1, 2, … and **drops** the old index (does not add it as a column). Use this when you just want a tidy index.

3. inplace=True – Change the existing DataFrame instead of creating a new one:

```
df_sub.reset_index(drop=True, inplace=True)
```

What it does: reset_index() normally **returns** a new DataFrame. With **inplace=True**, it updates the existing DataFrame (e.g. `df_sub`) and returns None. Use this when you want to modify the DataFrame you already have.

10. Sorting

Description: Sorting changes the order of rows by one or more columns. You can sort from low to high (ascending) or high to low (descending).

Syntax:

```
df.sort_values('col')                    # sort by one column (ascending)
df.sort_values('col', ascending=False)     # descending
df.sort_values(['col1','col2'], ascending=[True, False])  # by two columns
```

Example – sort by Salary (highest first):

Before: DataFrame `df` with rows in no special order.

Code:

```
df.sort_values('Salary', ascending=False)
```

After: Same rows, but ordered by Salary from highest to lowest (60000, 58000, 55000, 52000, 50000).

11. Missing Data (NaN)

Description: Missing values appear as **NaN** (Not a Number). You can find them with `isna()`, drop rows with `dropna()`, or replace them with `fillna(value)`.

Syntax:

```
df.isna()                    # True where missing (same: isnull())
df.isna().sum()              # count missing per column
df.dropna()                  # remove rows with any NaN
df.fillna(0)                 # replace NaN with 0
df['col'].fillna(df['col'].mean())  # fill with column mean
```

Example – fill missing Salary with the average salary:

Before: Some rows have NaN in the Salary column.

Code:

```
df['Salary'] = df['Salary'].fillna(df['Salary'].mean())
```

After: Every row has a number in Salary; previous NaN values are replaced by the mean of the column.

12. GroupBy and Aggregation

Description: GroupBy splits the table into groups (e.g. by Dept), then you apply a function (mean, sum, count) to each group. So you get one result per group instead of one per row. Single-column stats (mean, median) without grouping use `df['col'].mean()` etc.

Syntax:

```
df.groupby('col')['other_col'].mean()   # average per group
df.groupby('col').agg({'col1':'mean', 'col2':'sum'})  # different ops per column
```

Example – average Salary per Dept:

Before: DataFrame `df` with Dept = HR or IT and Salary for each person.

Code:

```
df.groupby('Dept')['Salary'].mean()
```

After: A Series: HR → 51000, IT → 57666.67 (average of salaries in each department).

Aggregates in Pandas – Calculating column statistics

Aggregate functions summarize many values in a column into a smaller set (e.g. one number or a short list). The general syntax is:

```
df.column_name.command()   or   df['column_name'].command()
```

Examples with our DataFrame:

```
df['Age'].median()      # 30  (middle value of 25, 28, 30, 32, 35)
df['Salary'].mean()     # 55000
df['Dept'].nunique()    # 2   (number of distinct values: HR, IT)
df['Dept'].unique()     # array(['HR', 'IT'], dtype=object)  — list of distinct values
df['Salary'].max()      # 60000
df['Age'].min()         # 25
df['Name'].count()      # 5   (number of non-null values)
```

Common column-statistic commands:

Command	Description
`mean`	Average of all values in the column
`std`	Standard deviation
`median`	Median (middle value)
`max`	Maximum value in the column
`min`	Minimum value in the column
`count`	Number of values in the column
`nunique`	Number of unique values in the column
`unique`	Array/list of unique values in the column

Use these when you need **one summary** for a whole column (e.g. "what's the median age?", "how many departments?", "what are the distinct departments?"). For summaries **by group**, use **groupby** (below).

Calculating aggregate functions over subsets

When we have a lot of data, we often want **aggregate statistics** (mean, median, standard deviation, percentiles, etc.) over **certain subsets** — for example, average grade per student, or average salary per department. Pandas gives us **.groupby** for this.

General syntax:

```
df.groupby('column1').column2.measurement()
```

- **column1** — the column to **group by** (defines the subsets)
- **column2** — the column to **measure** (the values we aggregate)
- **measurement** — the aggregate function (e.g. **mean**, **sum**, **median**, **max**, **min**, **count**)

Example – grade book: Suppose we have columns **student**, **assignment_name**, **grade**. DataFrame **before** groupby:

student	assignment_name	grade
Amy	Assignment 1	70
Amy	Assignment 2	90
Bob	Assignment 1	85
Bob	Assignment 2	95
Chris	Assignment 1	75

To get the **average grade per student**:

```
grades = df.groupby('student').grade.mean()
# student   grade
# Amy       80
# Bob       90
# Chris     75
```

With our DataFrame: We want **average salary per department**. Group by **Dept**, measure **Salary**, apply **mean**:

```
df.groupby('Dept')['Salary'].mean()
# Dept
# HR     51000
# IT     57666.67
```

You can write **df.groupby('Dept').Salary.mean()** (dot) or **df.groupby('Dept')['Salary'].mean()** (brackets); both work. **What it does:** Splits rows by **Dept**, then computes **mean** of **Salary** in each group.

Example 2 – multiple aggregates:

```
df.groupby('Dept')['Salary'].agg(['mean', 'sum', 'count'])
#         mean    sum  count
# Dept
# HR    51000  102000      2
# IT    57667  173000      3
```

Calculating aggregate functions II – cleaning groupby results

After **groupby**, the result is a **Series**, not a DataFrame: the **group values** are the **index**, and the aggregated values are the Series values. Often we want that index as a **column** in a proper DataFrame. Use **.reset_index()** right after the groupby:

```
df.groupby('column1').column2.measurement().reset_index()
```

Example – tea categories: Suppose we have a DataFrame **teas** with columns **id**, **tea**, **category**, **caffeine**, **price**. To get the **number of teas in each category**:

```
teas_counts = teas.groupby('category').id.count().reset_index()
#   category  id
# 0  black    3
# 1  green    4
# 2  herbal   8
```

The result column is called **id** because we counted the **id** column. To give it a clearer name, **rename** it:

```
teas_counts = teas_counts.rename(columns={"id": "counts"})
#   category  counts
# 0  black    3
# 1  green    4
# 2  herbal   8
```

With our DataFrame: Get average salary per department as a **DataFrame** (Dept as a column), then optionally rename the salary column:

```
dept_salary = df.groupby('Dept')['Salary'].mean().reset_index()
#   Dept   Salary
# 0  HR    51000
# 1  IT    57666.67

dept_salary = dept_salary.rename(columns={'Salary': 'avg_salary'})
#   Dept   avg_salary
# 0  HR    51000
# 1  IT    57666.67
```

Or **count employees per department** and rename the count column:

```
dept_counts = df.groupby('Dept')['Name'].count().reset_index()
dept_counts = dept_counts.rename(columns={'Name': 'employee_count'})
#   Dept   employee_count
# 0  HR    2
# 1  IT    3
```

Summary: Use **.reset_index()** after groupby to get a DataFrame with the group as a column; use **.rename(columns={...})** when the result column name (e.g. the column you measured) should be something clearer (e.g. **counts**, **avg_salary**).

Example 3 – different aggregates per column:

```
df.groupby('Dept').agg({
    'Salary': ['mean', 'sum'],
    'Age': 'max'
})
```

Example 4 – group by multiple columns:

```
df.groupby(['Dept', 'Age']).size()
```

Calculating aggregate functions IV – group by more than one column

Sometimes we want to group by **more than one column** (e.g. by store **and** by day of week). Pass a **list** of column names to **groupby**: **df.groupby(['col1', 'col2'])**. Pandas then forms one group for each **combination** of values (e.g. Chelsea + Monday, Chelsea + Tuesday, West Village + Monday, …).

Lecture example – average sales per location and day of week: Imagine a chain of stores. We have:

Location	Date	Day of Week	Total Sales
West Village	February 1	W	400
West Village	February 2	Th	450
Chelsea	February 1	W	375
Chelsea	February 2	Th	390
…	…	…	…

We want to see if sales differ by **location** and by **day of week**. So we take the **average** of **Total Sales** for each **(Location, Day of Week)** pair (e.g. “average sales at Chelsea on Mondays” across all dates):

```
df.groupby(['Location', 'Day of Week'])['Total Sales'].mean().reset_index()
```

Result (example):

Location	Day of Week	Total Sales
Chelsea	M	402.50
Chelsea	Tu	422.75
Chelsea	W	452.00
…	…	…
West Village	M	390
West Village	Tu	400

Location	Day of Week	Total Sales
…	…	…

So we get one row per **(Location, Day of Week)** with the average sales in that combination.

With our DataFrame: Average **Salary** for each **(Dept, Age)** combination (each department–age pair):

```
df.groupby(['Dept', 'Age'])['Salary'].mean().reset_index()
#   Dept  Age   Salary
# 0  HR   25    50000   (only Alice)
# 1  HR   28    52000   (only Diana)
# 2  IT   30    60000   (only Bob)
# 3  IT   32    58000   (only Eve)
# 4  IT   35    55000   (only Charlie)
```

Summary: Use **df.groupby(['col1', 'col2'])** (and more columns in the list if needed) to group by **multiple columns**. Then use **.column.measurement()** (e.g. **.mean()**) and **.reset_index()** to get a tidy DataFrame with one row per group.

Pivot tables

After a **groupby** across **multiple columns**, we often want to **reorganize** the result: turn one of the group columns into **column headers** (e.g. days of the week as separate columns) and keep the other as **rows**. That reorganization is called **pivoting**; the new table is a **pivot table**.

Lecture example – from long to wide: We had store data and did:

```
unpivoted = df.groupby(['Location', 'Day of Week'])['Total Sales'].mean().reset_index()
```

That gives a **long** table:

Location	Day of Week	Total Sales
Chelsea	M	300
Chelsea	Tu	310
Chelsea	W	375
…	…	…
West Village	Th	450
West Village	F	390
…	…	…

For comparison (e.g. “Chelsea vs West Village by day”), it’s often clearer to have **days as columns** and **Location as rows**:

Location	M	Tu	W	Th	F	Sa	Su
Chelsea	300	310	375	390	300	150	175

Location	M	Tu	W	Th	F	Sa	Su
West Village	300	310	400	450	390	250	200

Pandas pivot syntax:

```
df.pivot(columns='ColumnToPivot', index='ColumnToBeRows', values='ColumnToBeValues')
```

- **columns** — the column whose **values become the new column names** (e.g. Day of Week → M, Tu, W, …).
- **index** — the column that stays as **rows** (e.g. Location).
- **values** — the column that fills the **cells** (e.g. Total Sales).

Code for the store example:

```
# First groupby (long table)
unpivoted = df.groupby(['Location', 'Day of Week'])['Total Sales'].mean().reset_index()
# Then pivot (wide table)
pivoted = unpivoted.pivot(
    columns='Day of Week',
    index='Location',
    values='Total Sales'
)
# Pivot output can have odd indexing; use reset_index() if you want Location as a normal column
pivoted = pivoted.reset_index()
```

With our DataFrame: Suppose we have a **long** table of average salary per **(Dept, Age)** (e.g. from groupby). We can pivot so **Dept** is rows and **Age** is columns (or the other way around). Example: first build the long table, then pivot.

```
# Long: one row per (Dept, Age) with mean Salary
long_df = df.groupby(['Dept', 'Age'])['Salary'].mean().reset_index()
#   Dept  Age   Salary
# 0  HR   25    50000
# 1  HR   28    52000
# 2  IT   30    60000
# 3  IT   32    58000
# 4  IT   35    55000

# Pivot: Dept as rows, Age as columns, values = Salary
wide_df = long_df.pivot(columns='Age', index='Dept', values='Salary').reset_index()
#   Dept    25      28      30      32      35
# 0  HR    50000   52000   NaN     NaN     NaN
# 1  IT    NaN     NaN     60000   58000   55000
```

Takeaway: Use **.pivot(columns=..., index=..., values=...)** to turn a long groupby-style table into a wide one (one group dimension → columns). Use **.reset_index()** after pivot if you want a normal DataFrame with the index as a column.

Calculating aggregate functions III – custom aggregates with apply + lambda

Sometimes you need a **custom** summary that isn't just **mean**, **count**, or **sum** — for example a **percentile**. In that case, use **.apply()** with a **lambda** inside **groupby**.

What does “lambda gets one argument: all values in that group” mean?

- **groupby('category')** splits the table into one **group per category** (e.g. one group for "design", one for "marketing", one for "product").
- **.wage** means “take the **wage** column from each group.”
- **.apply(lambda x: ...)** runs your lambda **once per group**. Each time, **x** is **all the wage values in that group only** (pandas passes them as a small Series).

Concrete example with the employee table above:

Group (category)	What **x** is (wages in that group)	You return
design	x = 17, 27, 23	e.g. np.percentile(x, 75) → 23
marketing	x = 33	np.percentile(x, 75) → 33
product	x = 39, 48	np.percentile(x, 75) → 48

So the **lambda’s job** is: “given **x** = this group’s values, return one number (or value) that summarizes them.” You can use **any** function of **x**: **np.percentile(x, 75)**, **x.max()**, **x.min()**, **len(x)**, or a custom formula — whatever you need for that group.

What is a percentile? The **75th percentile** (i.e., the point at which 75% of employees have a lower wage and 25% have a higher wage) is the value such that 75% of the data is **below** it and 25% is **above** it. So it’s “higher than most” in a simple sense. NumPy’s **np.percentile(values, 75)** computes this for a list or array of numbers.

Lecture example – 75th percentile wage per category: Suppose we have employees with columns **id**, **name**, **wage**, **category**. DataFrame **before** groupby:

id	name	wage	category
10131	Sarah Carney	39	product
14189	Heather Carey	17	design
15004	Gary Mercado	33	marketing
11204	Cora Copaz	27	design
12900	Kim Park	48	product
20561	Sam Lopez	23	design
…	…	…	…

We want the **75th percentile of wage** in each category (e.g. “how high does the top quarter of wages go in design vs marketing?”):

```
import numpy as np

high_earners = df.groupby('category').wage.apply(
    lambda x: np.percentile(x, 75)
).reset_index()
#   category  wage
# 0  design   23
# 1  marketing 35
# 2  product   48
```

What happens step by step: (1) **groupby('category')** splits rows by category. (2) For each group, **.wage** gives that group's wages. (3) **.apply(lambda x: np.percentile(x, 75))** runs the lambda on each group: **x** is the list of wages in that group; we return the 75th percentile. (4) **.reset_index()** turns the result into a nice DataFrame with **category** and **wage** as columns.

With our DataFrame: Get the **75th percentile of Salary** in each **Dept** (so we see "upper-quarter" salary per department):

```
import numpy as np

dept_75pct = df.groupby('Dept')['Salary'].apply(
    lambda x: np.percentile(x, 75)
).reset_index()
dept_75pct = dept_75pct.rename(columns={'Salary': 'salary_75pct'})
#   Dept   salary_75pct
# 0  HR    52000.0      (75th % of 50000, 52000)
# 1  IT    59000.0      (75th % of 55000, 58000, 60000)
```

Takeaway: When **mean/count/sum** aren't enough, use **groupby(...).column.apply(lambda x: your_function(x))** where **x** is all values in that group. Then add **.reset_index()** (and **.rename** if you want clearer column names).

13. Merge and Join

Description: Merge joins two tables into one by matching rows on a common column (e.g. customer_id). You choose which rows to keep: only matches (**inner**), all from first table (**left**), all from second (**right**), or all from both (**outer**). **Concat** stacks two tables with the same columns (rows one below the other).

Syntax:

```
pd.merge(left, right, on='col')           # inner (default): only matching rows
pd.merge(left, right, on='col', how='left')   # all left + match right
pd.merge(left, right, on='col', how='outer')  # all rows from both
pd.concat([df1, df2])                     # stack rows (same columns)
```

Example – merge sales and targets on month: See the "sales vs targets" example below (before: two tables; code: `pd.merge(sales, targets)` ; after: one table with month, revenue, target).

Introduction: working with multiple DataFrames

Related data is often stored in **several tables** instead of one big table. That way we avoid **repeating** the same information many times, which keeps data smaller and easier to maintain.

Example – e-commerce orders: We could put everything in one table: order_id, customer_id, customer_name, customer_address, customer_phone_number, product_id, product_description, product_price, quantity, timestamp. But then:

- The **same customer** (name, address, phone) would appear again on every order.
- The **same product** (description, price) would appear again for every order that includes it.
  The table would get big and messy.

So we **split** the data into three tables:

Table	Columns	Purpose
orders	order_id, customer_id, product_id, quantity, timestamp	One row per order (who, what, how many, when)
products	product_id, product_description, product_price	One row per product
customers	customer_id, customer_name, customer_address, customer_phone_number	One row per customer

Example data (like in the lecture):

orders:

order_id	customer_id	product_id	quantity	timestamp
1	2	3	1	2017-01-01
2	2	2	3	2017-01-01
3	3	1	1	2017-01-01
4	3	2	2	2017-02-01
5	3	3	3	2017-02-01
6	1	4	2	2017-03-01
7	1	1	1	2017-02-02
8	1	4	1	2017-02-02

products:

product_id	description	price
1	thing-a-ma-jig	5
2	whatcha-ma-call-it	10
3	doo-hickey	7
4	gizmo	3

customers:

customer_id	customer_name	address	phone_number
1	John Smith	123 Main St.	212-123-4567
2	Jane Doe	456 Park Ave.	949-867-5309
3	Joe Schmo	798 Broadway	112-358-1321

To answer questions like “what did John Smith order?” or “what’s the total value of order 1?”, we **combine** these tables using **merge** (like SQL JOIN). The rest of this section covers the Pandas commands for that.

With our DataFrame: In this guide we use one table **df** (Name, Age, Salary, Dept). In real projects you might have **employees**, **departments**, and **projects** in separate tables and **merge** them when you need combined information (e.g. employee name with department name).

Inner Merge I – why we merge (matching rows)

From the **orders** table alone we can’t see the full picture: we see **customer_id** and **product_id**, but not the customer’s name or the product’s description and price. To get a **complete picture**, we **match** each order row with the corresponding row in **customers** (using customer_id) and in **products** (using product_id).

Example – order_id 1: In **orders**, order 1 has **customer_id** 2 and **product_id** 3. To find out who bought it, we look in **customers** for the row where **customer_id** = 2: that’s **Jane Doe**, 456 Park Ave., 949-867-5309. To find out what they bought, we look in **products** for **product_id** = 3: **doo-hickey**, $7. So order 1 = Jane Doe bought 1 doo-hickey.

Doing this kind of **matching** — linking rows from one table to rows in another using a common column (e.g. customer_id, product_id) — is called **merging** two DataFrames. Pandas **merge** does this for us in one step instead of looking up each row by hand.

With our DataFrame: If we had **orders** (with employee_id) and **df** as an **employees** table (with Name, Age, Salary, Dept), merging on employee_id would attach each order to the right employee (name, department, etc.). The next part shows the actual **merge** syntax.

merge – like SQL JOIN:

Inner Merge II – how .merge() works

Matching one row by hand is easy; matching **many rows** is tedious. Pandas does it for the whole table with **.merge()**.

The **.merge()** method (or **pd.merge()**):

1. Looks for **columns that appear in both** DataFrames (e.g. **customer_id** in orders and in customers).
2. Finds **rows where the values in those columns are the same** (e.g. order row with customer_id 2 and customer row with customer_id 2).
3. **Combines** each pair of matching rows into **one row** in a new table (order columns + customer columns side by side).

Syntax: `new_df = pd.merge(left_table, right_table)` . If both tables have a column with the same name (e.g. **customer_id**), Pandas uses it to match. You can also say **on='column_name'** to be explicit.

Lecture example – orders + customers:

```
new_df = pd.merge(orders, customers)
```

This matches **all** orders to the right customer: each order row is combined with that customer's row (customer_name, address, phone_number) so the new table has order_id, customer_id, product_id, quantity, timestamp **and** customer_name, address, phone_number.

Inner Merge III – DataFrame.merge() and chaining

Besides **pd.merge(left, right)**, every DataFrame has its own **.merge()** method. So you can write:

```
new_df = orders.merge(customers)
```

This gives the **same** result as **pd.merge(orders, customers)**. The DataFrame method is handy when you need to merge **more than two** tables: you can **chain** the calls. For example, merge **orders** with **customers**, then merge that result with **products** (so each order row gets customer info and product info):

```
big_df = orders.merge(customers).merge(products)
```

With our DataFrame: If we had **orders** (with customer_id and product_id), **customers**, and **products**, we could do **orders.merge(customers).merge(products)** to get one wide table. If we had **df** (employees) and two other tables — e.g. **departments** and **projects** — we could chain **df.merge(departments, on='Dept').merge(project_assignments, on='emp_id')** (with the right column names) to attach department and project info to each employee in one go.

With our DataFrame: If we had a **departments** table (e.g. Dept, Manager, Budget) and our **df** (Name, Age, Salary, Dept), we could attach department info to each employee:

```
employee_with_dept = pd.merge(df, departments, on='Dept')
# Each row gets Name, Age, Salary, Dept, plus Manager and Budget from the matching department row
```

Simple example (same as below):

```
left = pd.DataFrame({'id': [1, 2], 'name': ['A', 'B']})
right = pd.DataFrame({'id': [1, 2], 'score': [90, 85]})

pd.merge(left, right, on='id')
#   id name  score
# 0  1    A     90
# 1  2    B     85
```

Example – sales vs targets (merge on month):

Before merging – two separate tables:

sales:

month	revenue
January	300
February	290

month	revenue
March	310
April	325
May	475
June	495

targets:

month	target
January	310
February	270
March	300
April	350
May	475
June	500

Merge them (match on the common column `month`):

```
sales = pd.DataFrame({
    'month': ['January', 'February', 'March', 'April', 'May', 'June'],
    'revenue': [300, 290, 310, 325, 475, 495]
})
targets = pd.DataFrame({
    'month': ['January', 'February', 'March', 'April', 'May', 'June'],
    'target': [310, 270, 300, 350, 475, 500]
})
sales_vs_targets = pd.merge(sales, targets)
print(sales_vs_targets)
```

Result: Pandas matches rows where **month** is the same and combines them into one row per month:

month	revenue	target
January	300	310
February	290	270
March	310	300
April	325	350
May	475	475
June	495	500

Types: `how='inner'` (default), `'left'`, `'right'`, `'outer'`.

Example – left merge:

```
pd.merge(left, right, on='id', how='left')
# All rows from left; match from right where possible, else NaN
```

Outer merge – keep all rows from both tables

When we merge two DataFrames whose rows **don't match perfectly**, the **default (inner merge)** keeps only rows that match in both tables, so we **lose** unmatched rows. Sometimes we want to **keep** every row from both tables and just fill in missing values. That is an **outer merge** (**how='outer'**).

Example – Company A and Company B merge: Two companies have just merged. Each has a customer list with slightly different data. Company A has **name** and **email**; Company B has **name** and **phone**. Some customers appear in both, some only in one.

company_a:

name	email
Sally Sparrow	sally.sparrow@gmail.com
Peter Grant	pgrant@yahoo.com
Leslie May	leslie_may@gmail.com

company_b:

name	phone
Peter Grant	212-345-6789
Leslie May	626-987-6543
Aaron Burr	303-456-7891

If we used an **inner** merge, we would only get Peter Grant and Leslie May (the ones in both). We would **lose** Sally Sparrow (only in A) and Aaron Burr (only in B). An **outer join** includes **all rows from both tables**; where there is no match, the missing values are filled with **NaN** (Not a Number):

```
pd.merge(company_a, company_b, how='outer')
```

Result:

name	email	phone
Sally Sparrow	sally.sparrow@gmail.com	nan
Peter Grant	pgrant@yahoo.com	212-345-6789
Leslie May	leslie_may@gmail.com	626-987-6543
Aaron Burr	nan	303-456-7891

So we keep everyone: Sally has email, no phone; Peter and Leslie have both; Aaron has phone, no email.

Left and right merge

Using the same **company_a** and **company_b** tables above:

Left merge (how='left'): Keeps **all rows from the first (left) table**, and only rows from the second (right) table that **match** the left. The **order of the arguments matters**. If we put **company_a** first and use **how='left'**, we get **all** customers from Company A, and only add phone from Company B where that customer exists in B. So we get everyone who has email; missing phone is filled with **None**/NaN. Useful when we want "all from the left, plus matching info from the right" (e.g. "which customers have email but no phone?").

```
pd.merge(company_a, company_b, how='left')
```

Result:

name	email	phone
Sally Sparrow	sally.sparrow@gmail.com	None
Peter Grant	pgrant@yahoo.com	212-345-6789
Leslie May	leslie_may@gmail.com	626-987-6543

Right merge (how='right'): The opposite of left. Keeps **all rows from the second (right) table**, and only rows from the first (left) table that match the right. With **company_a** first and **company_b** second, **how='right'** gives **all** customers from Company B, and only adds email from Company A where that customer exists in A. So we get everyone who has phone; missing email is None/NaN. Useful for "all from the right, plus matching info from the left" (e.g. "which customers have phone but no email?").

```
pd.merge(company_a, company_b, how='right')
```

Result:

name	email	phone
Peter Grant	pgrant@yahoo.com	212-345-6789
Leslie May	leslie_may@gmail.com	626-987-6543
Aaron Burr	None	303-456-7891

Join on different column names:

```
pd.merge(left, right, left_on='id', right_on='emp_id')
```

Merge on specific columns – when column names don't line up

Sometimes the two tables don't have the **same column name** for the key we want to match on. We have to tell Pandas **which column in the left table** matches **which column in the right table** (e.g. **orders.product_id** with **products.id**).

Example – orders and products (two tables):

orders (order id, product_id, customer_id, quantity, timestamp):

id	product_id	customer_id	quantity	timestamp
1	3	2	1	2017-01-01
2	2	2	3	2017-01-01
3	1	3	1	2017-01-01
4	2	3	2	2016-02-01
5	3	3	3	2017-02-01
6	4	1	2	2017-03-01
7	1	1	1	2017-02-02
8	4	1	1	2017-02-02

products (product id, description, price):

id	description	price
1	thing-a-ma-jig	5
2	whatcha-ma-call-it	10
3	doo-hickey	7
4	gizmo	3

orders has **product_id**; **products** has **id**. The values match (1–4), but the **column names** are different. So we need a way to tell Pandas to match on these two columns.

One way to fix this: use .rename() so both tables have a **common column** for the merge. Rename the column **id** to **product_id** in **products**; then **orders** and **products** both have **product_id** and we can merge on it:

```
pd.merge(orders, products.rename(columns={'id': 'product_id'}))
```

Pandas will match rows where **product_id** is the same. The result has order columns plus **description** and **price** for that product. (You can store it: `orders_with_products = pd.merge(orders, products.rename(columns={'id': 'product_id'}))` .)

Other approach – left_on and right_on: Instead of renaming, say explicitly which column in each table to match:

```
orders_with_products = pd.merge(orders, products, left_on='product_id', right_on='id')
```

When both tables use "id" for different things: If **customers** and **products** both had a column called **id** (customer id vs product id), a default merge would be wrong. For **orders** + **customers** we'd use **left_on='customer_id', right_on='id'** or **customers.rename(columns={'id': 'customer_id'})** and merge on **customer_id**.

With our DataFrame: If **df** has **Dept** and a **departments** table has **id** and **name**, we could **pd.merge(df, departments, left_on='Dept', right_on='name')** or **departments.rename(columns={'id': 'Dept'})** and **pd.merge(df, departments, on='Dept')**.

Merge on specific columns II – left_on, right_on, and suffixes

Before merging – two tables:

orders (id = order id; we will match **customer_id** to customers):

id	product_id	customer_id	quantity	timestamp
1	3	2	1	2017-01-01
2	2	2	3	2017-01-01
3	1	3	1	2017-01-01
4	2	3	2	2016-02-01
5	3	3	3	2017-02-01
6	4	1	2	2017-03-01
7	1	1	1	2017-02-02
8	4	1	1	2017-02-02

customers (id = customer id; we will match **id** to orders.customer_id):

id	customer_name	address	phone_number
1	John Smith	123 Main St.	212-123-4567
2	Jane Doe	456 Park Ave.	949-867-5309
3	Joe Schmo	798 Broadway	112-358-1321

If we don't want to use **.rename()**, we can use **left_on** and **right_on** to say which column in the **left** table (the first one) and which in the **right** table (the second one) to match on. Here we match **customer_id** in **orders** to **id** in **customers**:

```
pd.merge(
    orders,
    customers,
    left_on='customer_id',
    right_on='id'
)
```

Duplicate column names – id_x and id_y: Both **orders** and **customers** may have a column named **id** (order id and customer id). Pandas does not allow two columns with the same name, so it renames them to **id_x** (from the left table) and **id_y** (from the right table). The result might look like:

id_x	customer_id	product_id	quantity	timestamp	id_y	customer_name	address	phone_number
1	2	3	1	2017-01-01 00:00:00	2	Jane Doe	456 Park Ave	949-867-5309
2	2	2	3	2017-01-01	2	Jane Doe	456 Park	949-867-5309

id_x	customer_id	product_id	quantity	timestamp	id_y	customer_name	address	phone_number
				00:00:00			Ave	
…	…	…	…	…	…	…	…	…

Using suffixes for clearer names: The names **id_x** and **id_y** are not very helpful. You can pass **suffixes** to use instead of **_x** and **_y**:

```
pd.merge(
    orders,
    customers,
    left_on='customer_id',
    right_on='id',
    suffixes=['_order', '_customer']
)
```

Then the columns become **id_order** (from orders) and **id_customer** (from customers), which makes the table easier to read.

With our DataFrame: If we merge **df** (employees) with a **departments** table that also has an **id** column, we might get **id_x** and **id_y**. To get clearer names: **pd.merge(df, departments, left_on='Dept', right_on='name', suffixes=['_emp', '_dept'])** so any overlapping column names become **col_emp** and **col_dept**.

Concatenate DataFrames

Sometimes a dataset is split into **multiple tables** (e.g. several CSV files so each download is smaller). To **reconstruct one DataFrame** from multiple smaller ones, use **pd.concat([df1, df2, df3, ...])**. This works when **all DataFrames have the same columns** (same names and meaning); concat then stacks the rows (or, with **axis=1**, the columns).

Example – two DataFrames with the same columns (name, email):

df1:

name	email
Katja Obinger	k.obinger@gmail.com
Alison Hendrix	alisonH@yahoo.com
Cosima Niehaus	cosi.niehaus@gmail.com
Rachel Duncan	rachelduncan@hotmail.com

df2:

name	email
Jean Gray	jgray@netscape.net
Scott Summers	ssummers@gmail.com
Kitty Pryde	kitkat@gmail.com
Charles Xavier	cxavier@hotmail.com

Combine them (stack rows):

```
pd.concat([df1, df2])
```

Result: One DataFrame with all 8 rows and the same columns (name, email):

name	email
Katja Obinger	k.obinger@gmail.com
Alison Hendrix	alisonH@yahoo.com
Cosima Niehaus	cosi.niehaus@gmail.com
Rachel Duncan	rachelduncan@hotmail.com
Jean Gray	jgray@netscape.net
Scott Summers	ssummers@gmail.com
Kitty Pryde	kitkat@gmail.com
Charles Xavier	cxavier@hotmail.com

axis: axis=0 (default) stacks **rows** (one table below the other). **axis=1** stacks **columns** (tables side by side). Use **axis=0** when the tables have the same columns and you want to append rows.

14. Dates and Time

Description: Date columns are often stored as text. Convert them to **datetime** so you can sort, filter, and extract year/month/day. Use `pd.to_datetime()` then the `.dt` accessor.

Syntax:

```
df['date'] = pd.to_datetime(df['date'])   # convert to datetime
df['year'] = df['date'].dt.year           # get year
df['month'] = df['date'].dt.month         # get month
df['day'] = df['date'].dt.day             # get day
```

Example – convert and extract year:

Before: A column `date` with strings like '2024-01-15'.

Code:

```
df['date'] = pd.to_datetime(df['date'])
df['year'] = df['date'].dt.year
```

After: `date` is datetime; `year` is a number (e.g. 2024). You can now filter by year or group by month.

15. Unique, Value Counts, Rename, Reset Index

Description: unique() gives the list of different values in a column (no repeats). **value_counts()** counts how many times each value appears. Rename and reset_index are in Section 9.

Syntax:

```
df['col'].unique()       # array of distinct values
df['col'].value_counts() # how many times each value appears
```

Example – unique departments and their counts:

Before: DataFrame `df` with a column Dept (HR, IT, IT, HR, IT).

Code:

```
df['Dept'].unique()       # ['HR', 'IT']
df['Dept'].value_counts() # IT 3, HR 2
```

After: `unique()` returns the two department names. `value_counts()` returns a Series: IT → 3, HR → 2.

Rename columns and reset index are covered in Section 9 – Adding and Dropping Columns (see **Rename columns** and **Reset index**).

16. Useful Methods Cheat Sheet

#	Category	Command / Code Example	What it does
1	Import	`import pandas as pd`	Import pandas; `pd` is the short name.
2	Create DataFrame	`df = pd.DataFrame({'name':['A','B'], 'age':[25,30]})`	Build a table from a dict, list, or list of lists.
3	Read CSV	`df = pd.read_csv('sales.csv')`	Load a CSV file into a DataFrame.
4	Read Excel	`df = pd.read_excel('data.xlsx', sheet_name='Sheet1')`	Read an Excel file; you can specify the sheet.
5	First few rows	`df.head(10)`	Show the first 10 rows (default is 5).
6	Last few rows	`df.tail()`	Show the last 5 rows.

#	Category	Command / Code Example	What it does
7	DataFrame size	`df.shape`	Returns (rows, columns).
8	Full summary	`df.info()`	Column names, types, and non-null counts.
9	Numerical summary	`df.describe()`	Mean, min, max, std, quartiles for numeric columns.
10	Column names	`df.columns`	List of all column names.
11	Column types	`df.dtypes`	Data type of each column (int, float, object, etc.).
12	Select 1 column	`df['salary']` or `df.salary`	Get one column as a Series.
13	Select multiple columns	`df[['name', 'age', 'city']]`	Select several columns; returns a DataFrame.
14	Filter rows	`df[df['age'] > 25]`	Keep only rows where the condition is True (like SQL WHERE).
15	Multiple conditions	`df[(df['salary'] > 60000) & (df['age'] < 30)]`	AND: both must be true (use `|` for OR).
16	Is in list	`df[df['city'].isin(['Delhi', 'Mumbai'])]`	Keep rows where the column value is in the list.
17	Sort by 1 column	`df.sort_values('salary', ascending=False)`	Sort by one column (ascending or descending).
18	Sort by multiple	`df.sort_values(['department', 'salary'], ascending=[True, False])`	Sort by department first, then salary.
19	New column (simple)	`df['bonus'] = df['salary'] * 0.12`	Add a column from a calculation.
20	New column (condition)	`df['status'] = df['age'].apply(lambda x: 'Senior' if x >= 30 else 'Junior')`	Add a column using apply + lambda for if/else logic.

#	Category	Command / Code Example	What it does
21	Drop column	`df.drop('bonus', axis=1, inplace=False)`	Remove a column (axis=1); axis=0 for rows.
22	Drop row	`df.drop(5, axis=0)`	Remove row with index 5.
23	Missing values count	`df.isnull().sum()`	Count of NaN per column (same as `df.isna().sum()`).
24	Remove missing rows	`df.dropna()`	Drop rows that have any missing value.
25	Fill missing	`df['salary'].fillna(0)` or `df.fillna(df.mean(numeric_only=True))`	Replace NaN with 0, mean, or other value.
26	Unique values	`df['department'].unique()`	Array of distinct values in the column.
27	Count of each value	`df['department'].value_counts()`	Frequency of each value (how many times each appears).
28	Group by + mean	`df.groupby('department')['salary'].mean()`	Average salary per department.
29	Group by + multiple agg	`df.groupby('city').agg({'salary': ['mean','max','count'], 'age':'mean'})`	Several aggregates at once (mean, max, count, etc.).
30	Merge two tables	`pd.merge(df1, df2, on='emp_id', how='left')`	Join two DataFrames on a common column (inner/left/right/outer).
31	Convert to date	`df['date'] = pd.to_datetime(df['date'])`	Convert string column to datetime.
32	Extract year/month	`df['year'] = df['date'].dt.year` and `df['month'] = df['date'].dt.month`	Get year and month from a datetime column.
33	Save to CSV	`df.to_csv('clean_data.csv', index=False)`	Write DataFrame to CSV (index=False skips row numbers).
34	Rename column	`df.rename(columns={'old_name':'new_name'}, inplace=True)`	Change column name(s).
35	Reset index	`df.reset_index(drop=True)`	Replace index with 0, 1, 2, … (drop=True discards old index).

17. Practice Set (with Solutions)

Use the same dataset for all questions. First run the code below to create **df** and **df2**, then try each step on your own. After you finish, check the solutions to cross-verify.

Dataset to create (copy this and run it once):

```
import pandas as pd

df = pd.DataFrame({
    'emp_id': [1, 2, 3, 4, 5, 6],
    'name': ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve', 'Frank'],
    'age': [25, 30, 35, 28, 32, 29],
    'salary': [50000, 60000, 55000, 52000, 58000, 54000],
    'dept': ['HR', 'IT', 'IT', 'HR', 'IT', 'Sales'],
    'join_date': ['2022-01-15', '2021-06-20', '2023-03-10', '2022-11-01', '2021-09-15', '2023-07-22']
})

df2 = pd.DataFrame({
    'emp_id': [1, 2, 3, 4, 5, 6],
    'rating': [4, 5, 4, 3, 5, 4]
})
```

Step-by-step questions

Q1. Display the first 3 rows of `df`.

Q2. Display the last 2 rows of `df`.

Q3. What is the shape of `df`? (How many rows and columns?)

Q4. Get the column names of `df`.

Q5. Select only the columns `name`, `age`, and `salary` from `df`.

Q6. Filter rows where `age` is greater than 28.

Q7. Filter rows where `dept` is **exactly** `'IT'`.

Q8. Filter rows where `dept` is either `'HR'` or `'IT'` (use `isin`).

Q9. Filter rows where `salary` is greater than 52000 **and** `age` is less than 32.

Q10. Sort `df` by `salary` in **descending** order (highest first).

Q11. Sort `df` by `dept` ascending, then by `salary` descending within each dept.

Q12. Add a new column `bonus` where bonus = 10% of salary (i.e. salary * 0.1).

Q13. Add a new column `level` where if age >= 30 then `'Senior'` else `'Junior'` (use `apply` and `lambda`).

Q14. Drop the column `bonus` from `df` (do not use `inplace`, so assign result to a new variable or overwrite `df`).

Q15. How many missing values are there in each column of `df`? (Use `isnull().sum()`.)

Q16. Get all **unique** values in the `dept` column.

Q17. Get the **count of each value** in the `dept` column (frequency).

Q18. What is the **average salary** per department? (Use `groupby` and `mean`.)

Q19. Per department, get both the **mean** and **max** of `salary` and the **count** of rows. (Use `groupby` and `agg`.)

Q20. Merge `df` and `df2` on `emp_id` so that all rows of `df` are kept (left merge). Store result in a variable like `merged`.

Q21. Convert the `join_date` column in `df` to datetime using `pd.to_datetime`.

Q22. From `join_date`, create two new columns: `join_year` and `join_month` (use `.dt.year` and `.dt.month`).

Q23. Rename the column `dept` to `department` in `df`.

Q24. Reset the index of `df` to 0, 1, 2, … and drop the old index.

Q25. Save `df` to a CSV file named `employees_clean.csv` without writing the row index.

Solutions (cross-verify with your answers)

A1. First 3 rows:

```
df.head(3)
```

A2. Last 2 rows:

```
df.tail(2)
```

A3. Shape (rows, columns):

```
df.shape   # (6, 6)
```

A4. Column names:

```
df.columns
```

A5. Select columns name, age, salary:

```
df[['name', 'age', 'salary']]
```

A6. Rows where age > 28:

```
df[df['age'] > 28]
```

A7. Rows where dept is 'IT':

```
df[df['dept'] == 'IT']
```

A8. Rows where dept is 'HR' or 'IT':

```
df[df['dept'].isin(['HR', 'IT'])]
```

A9. salary > 52000 AND age < 32:

```
df[(df['salary'] > 52000) & (df['age'] < 32)]
```

A10. Sort by salary descending:

```
df.sort_values('salary', ascending=False)
```

A11. Sort by dept then salary:

```
df.sort_values(['dept', 'salary'], ascending=[True, False])
```

A12. Add bonus column (10% of salary):

```
df['bonus'] = df['salary'] * 0.1
```

A13. Add level column (Senior/Junior by age):

```
df['level'] = df['age'].apply(lambda x: 'Senior' if x >= 30 else 'Junior')
```

A14. Drop column bonus:

```
df = df.drop('bonus', axis=1)
# or: df.drop(columns=['bonus'], inplace=True)
```

A15. Missing values per column:

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

A16. Unique values in dept:

```
df['dept'].unique()
```

A17. Count of each value in dept:

```
df['dept'].value_counts()
```

A18. Average salary per department:

```
df.groupby('dept')['salary'].mean()
```

A19. Per dept: mean and max of salary, count of rows:

```
df.groupby('dept').agg({'salary': ['mean', 'max', 'count']})
# or with count as length: df.groupby('dept')['salary'].agg(['mean', 'max', 'count'])
```

A20. Left merge df and df2 on emp_id:

```
merged = pd.merge(df, df2, on='emp_id', how='left')
```

A21. Convert join_date to datetime:

```
df['join_date'] = pd.to_datetime(df['join_date'])
```

A22. Extract year and month from join_date:

```
df['join_year'] = df['join_date'].dt.year
df['join_month'] = df['join_date'].dt.month
```

A23. Rename dept to department:

```
df.rename(columns={'dept': 'department'}, inplace=True)
```

A24. Reset index:

```
df = df.reset_index(drop=True)
```

A25. Save to CSV without index:

```
df.to_csv('employees_clean.csv', index=False)
```

Tip: Do the questions in order. If a later question assumes a new column (e.g. bonus) or a renamed column (e.g. department), you can re-run the dataset cell and then run only the steps you need up to that question, or keep one notebook where you run commands in sequence.

Happy learning!