**IMPORTANT QUESTION BASED ON INTERVIEW**

**1. How would you handle missing values in a Pandas DataFrame? (various strategies)**

**Answer:**

* **Detection:** df.isnull().sum()
* **Removal:**
  + df.dropna() → drop rows with NaN
  + df.dropna(axis=1) → drop columns with NaN
* **Imputation:**
  + df['col'].fillna(value) → fill with constant
  + df['col'].fillna(df['col'].mean()) → mean
  + df['col'].fillna(df['col'].median()) → median
  + df['col'].fillna(df['col'].mode()[0]) → mode
* **Forward/Backward fill:**
  + df.fillna(method='ffill') / df.fillna(method='bfill')
* **Advanced:** use sklearn.impute (KNN, IterativeImputer)

**2. Difference between apply(), map(), and applymap() in Pandas**

**Answer:**

* map() → element-wise operation **on Series only**.
* s.map(lambda x: x\*2)
* apply() → applies function along an **axis of DataFrame** (rows or columns) or to a Series.
* df['col'].apply(lambda x: x\*\*2) # Series
* df.apply(np.sum, axis=0) # DataFrame
* applymap() → element-wise operation **on entire DataFrame**.
* df.applymap(lambda x: str(x).upper())

**3. If you have a DataFrame with millions of rows, how would you improve performance when filtering data?**

**Answer:**

* Use **vectorized operations** (avoid Python loops).
* Convert columns to **categorical** if limited unique values.
* Use **query()** for filtering (df.query("col > 100")) → faster parsing.
* Use **NumPy functions** instead of Pandas when possible.
* Reduce memory usage:
  + Convert int64 → int32 if values fit.
  + Convert object → category where applicable.
* Use **chunking** (pd.read\_csv(..., chunksize=100000)) for large files.
* For extreme cases, use **Dask** or **Polars**.

**4. What’s the difference between merge(), join(), and concat() in Pandas?**

**Answer:**

merge() → SQL-style joins on columns or indices.

pd.merge(df1, df2, on='id', how='inner')

join() → simpler syntax for joining **on index** (or key column if specified).

df1.join(df2, on='id', how='left')

concat() → stack DataFrames **vertically or horizontally**.

pd.concat([df1, df2], axis=0) # row-wise

pd.concat([df1, df2], axis=1) # column-wise

**5. How do you handle categorical data in Pandas efficiently?**

**Answer:**

* Convert to category type:
* df['col'] = df['col'].astype('category')
  + Saves memory.
  + Speeds up comparisons and groupby operations.
* Encode categories:
  + **Label Encoding:**
  + df['col'].cat.codes
  + **One-Hot Encoding:**
  + pd.get\_dummies(df['col'], drop\_first=True)
* For large datasets, keep categorical dtype instead of object strings.

**LOADING DATASET**

**1. If you have a 10GB CSV file but only 16GB RAM, how would you load and process it in Pandas?**

**Answer:**

Use **chunksize** parameter:

for chunk in pd.read\_csv("file.csv", chunksize=100000):

process(chunk)

Use **dtypes optimization** → downcast int64 → int32, float64 → float32, convert object → category.

Load only required columns with usecols=[...].

Use **compression** if file is gzipped (compression='gzip').

For very large data → use **Dask**, **Polars**, or database storage instead of full Pandas load.

**2. How do you load data from SQL into Pandas?**

**Answer:**

Use **read\_sql** (requires SQLAlchemy or DBAPI connection).

import pandas as pd

import sqlite3

conn = sqlite3.connect("mydb.db")

df = pd.read\_sql("SELECT \* FROM table\_name", conn)

Also works with pd.read\_sql\_query() or pd.read\_sql\_table().

For big tables → use chunksize to process incrementally.

**3. What is the difference between read\_csv() and read\_table()?**

**Answer:**

read\_csv() → defaults to **comma (,)** as delimiter.

pd.read\_csv("file.csv")

read\_table() → defaults to **tab (\t)** as delimiter.

pd.read\_table("file.txt")

Otherwise, both are functionally the same (you can override sep in either).

**4. How would you handle loading data from an API into a DataFrame?**

**Answer:**

Use **requests** (or similar library) to fetch JSON/CSV from API.

import requests

import pandas as pd

url = "https://api.example.com/data"

response = requests.get(url)

data = response.json() # if JSON

df = pd.DataFrame(data)

If API returns CSV:

from io import StringIO

df = pd.read\_csv(StringIO(response.text))

For paginated APIs → loop over pages, append DataFrames.

**Accessing Data from Data Frame**

Here’s the **interview-style Q&A cheat sheet** for your new set of questions 👇

**1. What’s the difference between df.loc[], df.iloc[], and df.at[]?**

**Answer:**

**df.loc[]** → label-based indexing (row/column names).

df.loc[5, "Salary"] # row with index label 5, column Salary

**df.iloc[]** → position-based indexing (row/column integer positions).

df.iloc[5, 2] # 6th row, 3rd column

**df.at[]** → fast scalar access (single value, label-based).

df.at[5, "Salary"] # faster than loc for single element

**2. How would you select multiple conditions in Pandas (e.g., employees with Age > 30 and Salary < 50,000)?**

**Answer:**

Use **bitwise operators & (and), | (or), ~ (not)** with parentheses.

df[(df["Age"] > 30) & (df["Salary"] < 50000)]

**3. How can you efficiently retrieve only a subset of columns from a large DataFrame?**

**Answer:**

Use usecols while loading:

df = pd.read\_csv("file.csv", usecols=["Name", "Salary"])

Or select directly:

df\_subset = df[["Name", "Salary"]]

Helps reduce memory usage significantly for wide DataFrames.

**4. How would you extract the top 5 highest-paid employees from a dataset?**

**Answer:**

Using nlargest():

df.nlargest(5, "Salary")

Or using sort\_values():

df.sort\_values("Salary", ascending=False).head(5)

Would you like me to **combine all Pandas interview Q&As so far** (missing values, apply/map/applymap, filtering, merge/join/concat, SQL, API, loc/iloc, etc.) into a **single well-structured guide**? That way you’ll have one go-to Pandas interview prep sheet.