

NUMPY & PANDAS

for Data Science Interviews



- 1. NumPy — Array Creation & Broadcasting
- 2. NumPy — Vectorized Ops, Axis Operations, Reshaping
- 3. Pandas — Series vs DataFrame, loc/iLoc
- 4. Pandas — GroupBy & Aggregation
- 5. Pandas — Merge / Join / Concat
- 6. Pandas — Pivot Table
- 7. Pandas — Missing Data
- 8. Pandas — apply / map / applymap
- 9. Pandas — Datetime Operations
- 10. Quick Reference — Which Function When

Phase 1 • Placement-Oriented Notes • Not Exam-Based

NumPy — Array Creation & Broadcasting

NumPy array = fixed-type, contiguous block of memory → enables fast vectorized math, unlike Python Lists (which store pointers to scattered objects).

Creating Arrays

```
import numpy as np

a = np.array([1, 2, 3])           # 1D array from list
z = np.zeros((3, 4))             # 3x4 array of zeros
o = np.ones((2, 2))             # 2x2 array of ones
r = np.arange(0, 10, 2)          # [0, 2, 4, 6, 8] -- like range()
lin = np.linspace(0, 1, 5)       # 5 evenly spaced values 0 to 1
rand = np.random.rand(3, 3)      # 3x3 random floats [0,1)
```

► Broadcasting Rules

Broadcasting Lets NumPy perform operations on arrays of different shapes without writing explicit loops or copying data.

```
a = np.array([[1, 2, 3],
              [4, 5, 6]])      # shape (2,3)
b = np.array([10, 20, 30])     # shape (3,)

print(a + b)
# [[11 22 33]
#  [14 25 36]]
# b is "stretched" across each row -- no actual copy is made (memory efficient)

# Broadcasting rule: compare shapes from the RIGHT.
# Dimensions are compatible if equal, OR one of them is 1.
# (2,3) vs (3,) -> (2,3) vs (1,3) -> compatible -> result (2,3)

col = np.array([[1], [2], [3]]) # shape (3,1)
row = np.array([10, 20, 30])    # shape (3,)
print(col + row)                # -> broadcasts to (3,3) grid
```



🔗 Common Interview Questions

- Why is NumPy faster than a Python list for math operations? → Contiguous fixed-type memory + vectorized C loops under the hood, avoiding Python-level per-element loop overhead.
- What is broadcasting and why does it matter? → Lets arrays of different (but compatible) shapes be combined without manually reshaping/copying — memory efficient and concise.
- When do two shapes fail to broadcast? → When, compared from the right, dimensions are neither equal nor equal to 1 (e.g. (2,3) and (4,) cannot broadcast).
- np.array vs np.asarray? → asarray won't copy if the input is already an ndarray of the right dtype; array always copies by default.

Vectorized Ops, Axis Operations & Reshaping

Vectorized Operations

Operations apply element-wise automatically — no Python-level for-loop needed. This is the core reason NumPy/Pandas code is fast.

```
a = np.array([1, 2, 3, 4])
print(a * 2)           # [2 4 6 8] -- no explicit loop needed
print(a > 2)           # [False False True True] -- boolean mask
print(a[a > 2])        # [3 4] -- boolean indexing / filtering
print(np.where(a > 2, "high", "low")) # conditional vectorized replace
```

► Axis — the #1 Confusion Point

axis=0 → operate down the rows (column-wise result). axis=1 → operate across the columns (row-wise result). Trick: axis=N means "the dimension that disappears / gets collapsed".

```
m = np.array([[1, 2, 3],
              [4, 5, 6]]) # shape (2,3)

print(m.sum())           # 21 -- sum of ALL elements
print(m.sum(axis=0))     # [5 7 9] -- sum DOWN each column (collapses rows)
print(m.sum(axis=1))     # [6 15] -- sum ACROSS each row (collapses columns)

print(m.reshape(3, 2))  # reshapes to 3 rows x 2 cols (same 6 elements)
print(m.reshape(-1))    # flattens to 1D: [1 2 3 4 5 6]
print(m.T)              # transpose -> shape (3,2)
```

🕒 Common Interview Questions

- What does axis=0 vs axis=1 mean in sum()/mean()? → axis=0 collapses rows (result per column), axis=1 collapses columns (result per row). Easy to mix up — always sanity check with a small example.
- What is boolean indexing and why is it useful? → Using a condition (arr > x) as an index to filter elements — vectorized, avoids explicit loops, used constantly in EDA filtering.
- reshape(-1) vs flatten() vs ravel()? → -1 lets NumPy auto-infer that dimension; flatten() always returns a copy, ravel() returns a view when possible (faster, but changes can affect original array).
- Difference between reshape and resize? → reshape returns a new view/array without changing original (errors if size mismatch); resize can change the array in-place and pad/truncate.

Series vs DataFrame, loc / iloc

Series → 1D labeled array (single column) + multiple Series sharing an index =

DataFrame → 2D labeled table

```
import pandas as pd

s = pd.Series([10, 20, 30], index=["a", "b", "c"]) # 1D, labeled
df = pd.DataFrame({
    "name": ["Amit", "Neha", "Ravi"],
    "score": [85, 92, 78]
}) # 2D, labeled rows+cols
```

► loc vs iloc

```
df = pd.DataFrame({"score": [85, 92, 78]}, index=["a", "b", "c"])

df.loc["b"] # label-based -> row with index label "b"
df.loc["a":"b"] # label slicing -> INCLUSIVE of "b"

df.iloc[1] # position-based -> 2nd row (0-indexed)
df.iloc[0:2] # position slicing -> EXCLUSIVE of index 2 (like Python)

df.loc[df["score"] > 80] # loc also supports boolean condition filtering
```

Aspect	.loc	.iloc
Selects by	Label / index name	Integer position
Slicing end	Inclusive	Exclusive (Python-style)
Boolean filtering	Yes — df.loc[condition]	Not directly

🕒 Common Interview Questions

- loc vs iloc — the key difference? → loc uses labels (and slicing is inclusive), iloc uses integer positions (slicing is exclusive, like normal Python).
- Why does df.loc['a':'b'] include 'b' but df.iloc[0:2] excludes index 2? → loc slicing is label-based and inclusive by design; iloc mirrors standard Python integer slicing.
- Series vs single-column DataFrame? → Series is 1D (df['col']), single-column DataFrame is 2D (df[['col']]) — different shape, affects downstream operations like sklearn expecting 2D input for features.

GroupBy & Aggregation

GroupBy follows the Split → Apply → Combine pattern: split data into groups by a key, apply a function (mean/sum/count...) to each group, combine results back into one output.

Split by key

Apply function per group

Combine into result

```
df = pd.DataFrame({
    "dept": ["Sales", "Sales", "Tech", "Tech", "Tech"],
    "salary": [50000, 55000, 70000, 72000, 68000],
    "bonus": [2000, 2500, 3000, 3200, 2800]
})

# Single aggregation
df.groupby("dept")["salary"].mean()
# dept
# Sales    52500.0
# Tech     70000.0

# Multiple aggregations at once
df.groupby("dept").agg(
    avg_salary=("salary", "mean"),
    max_bonus=("bonus", "max"),
    count=("salary", "size")
)

# Group + transform (keeps original row count, e.g. for feature engineering)
df["dept_avg_salary"] = df.groupby("dept")["salary"].transform("mean")
```

`.agg()` lets you compute multiple named aggregations in a single readable call — the practical, interview-preferred way over chaining separate `groupby` calls.

`.transform()` is key for feature engineering — it broadcasts the group result back to every original row (same length as input), unlike `.agg()` which collapses rows.

🕒 Common Interview Questions

- `groupby().agg()` vs `groupby().transform()`? → `agg()` reduces each group to one row (summary); `transform()` returns a result the same length as the original DataFrame, aligned back to each row (used to create features like 'dept average').
- How do you apply different aggregations to different columns? → `df.groupby('key').agg(new_col=('col', 'func'), ...)` — named aggregation syntax.
- How would you find, per group, the row with the max value of some column? → `df.loc[df.groupby('dept')['salary'].idxmax()]`

Merge / Join / Concat

```
orders = pd.DataFrame({"order_id": [1, 2, 3], "cust_id": [101, 102, 104]})
customers = pd.DataFrame({"cust_id": [101, 102, 103], "name": ["A", "B", "C"]})

pd.merge(orders, customers, on="cust_id", how="inner")  # only matching cust_id (2,103
dropped... wait)
# INNER -> only cust_id present in BOTH (101, 102)

pd.merge(orders, customers, on="cust_id", how="left")  # all rows from orders, NaN if
no match (104 -> NaN)
pd.merge(orders, customers, on="cust_id", how="right")  # all rows from customers
pd.merge(orders, customers, on="cust_id", how="outer")  # all rows from both, NaN where
missing

# .join() -- shortcut for merging on INDEX instead of a column
orders.set_index("cust_id").join(customers.set_index("cust_id"), how="left")

# concat -- stacks DataFrames, no key-matching logic at all
pd.concat([orders, orders], axis=0)  # stack rows (like UNION)
pd.concat([orders, customers], axis=1)  # stack columns side by side
```

► Join Types — Visual

INNER → common rows only

LEFT → all of left + matches

RIGHT → all of right + matches

OUTER → everything, NaN if missing

🕒 Common Interview Questions

- merge vs join vs concat — when to use which? → merge = SQL-style join on column(s) with full control over how; join = shortcut for merging on the INDEX; concat = simple stacking of DataFrames (rows or columns), no key matching.
- What happens to unmatched rows in a left join? → They're kept, with NaN filled in for columns coming from the right (unmatched) table.
- concat axis=0 vs axis=1? → axis=0 stacks vertically (adds more rows, like SQL UNION); axis=1 stacks horizontally (adds more columns, aligned by index).
- How do you merge on columns with different names in each DataFrame? → `pd.merge(df1, df2, left_on='colA', right_on='colB')`

Pivot Table

`pivot_table` = `groupby` + `reshape` combined — summarizes data into a cross-tab / spreadsheet-style matrix (rows x columns x aggregated values).

```
df = pd.DataFrame({
    "dept": ["Sales", "Sales", "Tech", "Tech"],
    "gender": ["M", "F", "M", "F"],
    "salary": [50000, 55000, 70000, 72000]
})

pd.pivot_table(df, values="salary", index="dept",
               columns="gender", aggfunc="mean")

#           F           M
# dept
# Sales  55000.0  50000.0
# Tech   72000.0  70000.0

pd.pivot_table(df, values="salary", index="dept", aggfunc=["mean", "count"])
```

Effectively equivalent to `df.groupby(["dept", "gender"])`

`["salary"].mean().unstack()` — `pivot_table` is just more readable and lets you pass multiple `aggfuncs` at once.

🔗 Common Interview Questions

- `pivot_table` vs `groupby` — what's the practical difference? → Same underlying idea (split-apply-combine), but `pivot_table` directly outputs a reshaped matrix (rows x columns), while `groupby` needs an extra `unstack()`/`reset_index()` to get the same shape.
- What does `pivot_table` do with duplicate index/column combos that `groupby` wouldn't need to worry about? → It aggregates them using `aggfunc` (default mean) — that's WHY an `aggfunc` is required, unlike plain `pivot()` which errors on duplicates.
- `pivot()` vs `pivot_table()`? → `pivot()` reshapes without aggregation (errors on duplicate index/column pairs); `pivot_table()` aggregates duplicates automatically.

Handling Missing Data

```
df.isna().sum()           # count missing values per column
df.dropna()              # drop rows with ANY NaN
df.dropna(subset=["salary"]) # drop rows only if 'salary' is NaN
df.dropna(thresh=3)       # keep rows with at least 3 non-NaN values

df["salary"].fillna(df["salary"].mean()) # impute with mean
df["dept"].fillna(df["dept"].mode()[0])  # impute categorical with mode
df.fillna(method="ffill")                # forward-fill (carry last value)
df.interpolate()                         # linear interpolation for numeric gaps
```

► Drop vs Impute — Decision Guide

Situation	Recommended approach
Very few rows missing (<5%)	Drop rows
Column mostly missing (>50%)	Drop column
Numeric, missing at random	Mean/median impute
Categorical	Mode impute or "Unknown" category
Time series	Forward-fill / interpolate

🔗 Common Interview Questions

- Mean vs median imputation — when to prefer median? → Median is robust to outliers/skewed distributions; mean is fine for roughly symmetric, outlier-free data.
- Why can dropping rows with missing values be risky? → If data isn't missing completely at random, dropping introduces bias (e.g. high-income respondents skipping the income field).
- ffill vs bfill vs interpolate for time series? → ffill carries the last known value forward, bfill carries the next value backward, interpolate estimates values linearly (or otherwise) between known points — interpolate is usually more accurate for continuous trends.

apply / map / applymap

```
s = pd.Series([1, 2, 3])
s.map(lambda x: x * 2)           # Series only -- element-wise

df = pd.DataFrame({"a": [1, 2], "b": [3, 4]})
df["a"].apply(lambda x: x * 2)   # element-wise on ONE column (Series)
df.apply(lambda col: col.sum())  # applies a function along an axis (per column/row)
df.apply(lambda row: row["a"] + row["b"], axis=1)  # row-wise, axis=1

df.applymap(lambda x: x * 2)      # element-wise on EVERY cell of the DataFrame
# (Pandas 2.1+: use df.map instead, applymap is deprecated)
```

Function	Works on	Granularity
.map()	Series only	Element-wise
.apply()	Series or DataFrame	Element-wise (Series) or row/column-wise (DataFrame, via axis)
.applymap()	DataFrame only	Element-wise on every cell

Prefer vectorized operations (e.g. `df['a'] * 2`) over `apply` wherever possible — `apply` is a hidden Python loop and is much slower on large datasets.

🔗 Common Interview Questions

- Why avoid `apply()` when a vectorized alternative exists? → `apply()` still loops row-by-row in Python under the hood — much slower than NumPy/Pandas' compiled vectorized operations, especially on large datasets.
- `map` vs `apply` on a Series — any difference? → `map` is meant for simple element-wise substitution/lookup (including via a dict), `apply` is more general and can take more complex functions.
- How do you apply a function row-wise across a DataFrame? → `df.apply(func, axis=1)` — but check if a vectorized version is possible first for performance.

Datetime Operations

```
df["date"] = pd.to_datetime(df["date"])    # parse strings into datetime64

df["year"] = df["date"].dt.year
df["month"] = df["date"].dt.month
df["weekday"] = df["date"].dt.day_name()    # 'Monday', 'Tuesday'...

df.set_index("date", inplace=True)
df.resample("M").mean()                    # monthly aggregation (like groupby for
time)

(df["end_date"] - df["start_date"]).dt.days    # difference between dates -> int days
```

`resample()` is groupby for time-indexed data — bucket rows into time periods (day/week/month) and aggregate.
Essential for any time-series feature engineering.

🕒 Common Interview Questions

- Why convert date strings to `pd.to_datetime` instead of keeping them as strings? → Enables all `.dt` accessor operations (year/month/weekday extraction), correct chronological sorting, and resampling — string dates sort lexically, not chronologically.
- groupby vs resample for time series? → resample is specifically for a `DatetimeIndex` and understands calendar periods (month-end, week start); groupby needs you to manually extract a period column first.
- How do you extract the day-of-week or check if a date is a weekend? → `df['date'].dt.day_name()` or `df['date'].dt.dayofweek >= 5` for weekend check.

Quick Reference — Which Function When

► Combining Data

Need	Use
Combine on a common column, SQL-style	<code>pd.merge()</code>
Combine on the index	<code>.join()</code>
Just stack rows or columns, no matching	<code>pd.concat()</code>

► Selecting Data

Need	Use
Select by label / index name	<code>.loc[]</code>
Select by integer position	<code>.iloc[]</code>
Filter rows by condition	<code>.loc[condition]</code> or <code>df[condition]</code>

► Summarizing Data

Need	Use
Group + summarize to fewer rows	<code>groupby().agg()</code>
Group + keep original row count	<code>groupby().transform()</code>
Rows x Columns cross-tab of aggregated values	<code>pivot_table()</code>
Reshape without aggregating (no duplicates)	<code>pivot()</code>

► Row/Element-wise Functions

Need	Use
Element-wise on a Series (incl. dict lookup)	<code>.map()</code>
Function on a Series, or row/col-wise on DataFrame	<code>.apply()</code>
Element-wise on every DataFrame cell	<code>.applymap()</code> (legacy) → <code>.map()</code>
Same operation on whole column/array, no custom func	Vectorized op (fastest)

► Missing Data

Need	Use
Remove missing rows/cols	<code>.dropna()</code>
Fill with a fixed/statistical value	<code>.fillna()</code>
Estimate based on surrounding trend	<code>.interpolate()</code>

General rule for interviews: always be ready to explain the trade-off behind your choice (e.g. "I used median over mean because the salary column was right-skewed with outliers").