Pandas Fundamentals

1. What Is pandas?

pandas is a Python library for working with structured and tabular data.

It is commonly used when your data looks like:

```
Name     Age   Score   City
Hafsa    25    91      Gujranwala
Asim     27    84      Lahore
Sara     24    88      Islamabad
```

pandas provides two core data structures:

- `Series` — one-dimensional labeled data
- `DataFrame` — two-dimensional tabular data with rows and columns

The library is built on top of NumPy and is designed for data manipulation and analysis.

The conventional import is:

```
import pandas as pd
```

---

2. Why Do We Need pandas?

Python lists and dictionaries can store data, and NumPy is excellent for numerical arrays.

But real-world datasets often need operations such as:

```
Load CSV
    □
Inspect data
    □
Select columns
    □
Filter rows
    □
Clean missing values
    □
Create new columns
    □
Group data
    □
Combine datasets
    □
Calculate statistics
    □
Export results
```

pandas is designed around this workflow.

The official pandas documentation specifically includes tools for selection, missing data, grouping, reshaping, combining datasets, and importing/exporting data.

---

3. The pandas Mental Model

Think of pandas like a spreadsheet controlled with Python.

```
        DataFrame
    □□□□□□□□□□□□□□□□□□□□□□
```

```
    │ Name │ Score │ Age│
    ├──────┼───────┼────┤
    │ Ali  │  85   │ 21 │
    │ Sara │  92   │ 23 │
    │ Asim │  78   │ 25 │
    └──────┴───────┴────┘
```

But unlike a spreadsheet, you can programmatically:

- process thousands or millions of rows
- repeat transformations
- combine datasets
- automate reports
- clean inconsistent data
- integrate with APIs and databases
- build reproducible analysis pipelines

---

4. Installing pandas

Install pandas with:

```
pip install pandas
```

Verify it:

```
import pandas as pd

print(pd.__version__)
```

The current pandas documentation lists pandas 3.0.6 as the current documentation version.

---

5. Series

A `Series` is a one-dimensional labeled data structure.

```
import pandas as pd

scores = pd.Series([85, 92, 78, 90])

print(scores)
```

Conceptually:

```
0   85
1   92
2   78
3   90
```

The left side is the **index**.

The right side contains the values.

A Series can also have meaningful labels:

```
scores = pd.Series(
    [85, 92, 78],
    index=["Ali", "Sara", "Asim"]
)
```

Now:

```
Ali    85
```

```
Sara    92
Asim    78
```

pandas defines a Series as a one-dimensional labeled array.

---

6. DataFrame

A `DataFrame` is the main pandas structure for tabular data.

```
students = pd.DataFrame({
    "name": ["Ali", "Sara", "Asim"],
    "score": [85, 92, 78],
    "age": [21, 23, 25]
})

print(students)
```

Result:

```
   name  score  age
0   Ali     85   21
1  Sara     92   23
2  Asim     78   25
```

A DataFrame can be thought of as a collection of Series sharing the same row index.

---

7. Creating a DataFrame From Lists

```
data = [
    ["Ali", 85, 21],
```

```
    ["Sara", 92, 23],
    ["Asim", 78, 25]
]

df = pd.DataFrame(
    data,
    columns=["name", "score", "age"]
)
```

This is useful when your data already exists as Python records.

---

8. Creating a DataFrame From Dictionaries

This is often easier to read:

```
df = pd.DataFrame({
    "name": ["Ali", "Sara", "Asim"],
    "score": [85, 92, 78],
    "age": [21, 23, 25]
})
```

Each dictionary key becomes a column.

Each list becomes the values for that column.

---

9. Inspecting Data

Before manipulating a dataset, inspect it.

```
df.head()
```

Shows the first rows.

```
df.tail()
```

Shows the last rows.

The pandas user guide includes `head()` and `tail()` among its essential data-inspection tools.

---

10. Understanding the Dataset

Use:

```
df.shape
```

Example:

```
(1000, 5)
```

Meaning:

```
1000 rows
5 columns
```

Check column names:

```
df.columns
```

Check index:

`df.index`

Check data types:

`df.dtypes`

Get a compact overview:

`df.info()`

---

11. describe()

For numerical columns:

`df.describe()`

It can provide statistics such as:

`count`
`mean`
`std`
`min`
`25%`
`50%`
`75%`
`max`

For example:

`print(df["score"].describe())`

This is useful for quickly understanding the distribution of numerical data.

---

12. Selecting a Column

Use the column name:

`df["score"]`

This returns a `Series`.

You can store it:

`scores = df["score"]`

You can select multiple columns:

`df[["name", "score"]]`

The result remains a DataFrame.

A single DataFrame column selected with `df["column"]` is a Series.

---

13. Selecting Rows With `loc`

`loc` selects using labels.

`df.loc[0]`

Select specific rows and columns:

`df.loc[0:2, ["name", "score"]]`

Select a single cell:

```
df.loc[0, "score"]
```

For production code, pandas recommends explicit access methods such as `loc`, `iloc`, `at`, and `iat` for data access.

---

14. Selecting Rows With `iloc`

`iloc` selects based on integer positions.

```
df.iloc[0]
```

First three rows:

```
df.iloc[0:3]
```

First row, second column:

```
df.iloc[0, 1]
```

First two rows and first two columns:

```
df.iloc[0:2, 0:2]
```

Mental model:

```
loc  → labels
iloc → positions
```

---

15. at and iat

When you need a single value:

```
df.at[0, "score"]
```

or:

```
df.iat[0, 1]
```

Think:

```
at  → label-based single value
iat → position-based single value
```

---

16. Filtering Rows

Suppose:

```
df = pd.DataFrame({
    "name": ["Ali", "Sara", "Asim", "Hina"],
    "score": [85, 92, 78, 95]
})
```

Find students with scores above 90:

```
high_scores = df[df["score"] > 90]
```

Result:

```
  name  score
```

```
1  Sara    92
3  Hina    95
```

The expression:

```
df["score"] > 90
```

creates a Boolean mask.

---

17. Multiple Conditions

Use parentheses:

```
result = df[
    (df["score"] >= 80) &
    (df["score"] <= 90)
]
```

For OR:

```
result = df[
    (df["score"] < 80) |
    (df["score"] > 90)
]
```

For NOT:

```
result = df[
    ~(df["score"] > 90)
]
```

Remember:

`&` → AND
`|` → OR
`~` → NOT

Do not use Python's `and` and `or` for element-wise pandas conditions.

---

18. `isin()`

Suppose you want specific cities:

```
df[df["city"].isin([
    "Lahore",
    "Islamabad"
])]
```

This is cleaner than writing multiple OR conditions.

You can also exclude values:

```
df[
    ~df["city"].isin(["Lahore"])
]
```

---

19. `query()`

pandas also provides `query()` for readable filtering.

```
df.query("score > 90")
```

Multiple conditions:

```
df.query(
    "score >= 80 and score <= 90"
)
```

`query()` can be convenient for readable expressions, while Boolean indexing remains a fundamental pandas technique.

---

20. Sorting Data

Sort by one column:

```
df.sort_values("score")
```

Descending:

```
df.sort_values(
    "score",
    ascending=False
)
```

Multiple columns:

```
df.sort_values(
    ["city", "score"],
    ascending=[True, False]
)
```

This means:

city □ ascending
score □ descending

---

21. Adding a Column

You can create a new column directly:

```
df["passed"] = df["score"] >= 60
```

Now:

```
   name  score  passed
0  Ali     85    True
1  Sara    92    True
2  Asim    78    True
```

This is one of the most useful pandas patterns.

---

22. Creating Derived Columns

Suppose:

```
df["score"] = [85, 92, 78]
```

Create a percentage:

```
df["percentage"] = df["score"] / 100 * 100
```

Or calculate a bonus:

```
df["bonus_score"] = df["score"] + 5
```

pandas allows operations on entire columns without manually looping through each row.

---

23. Conditional Columns With `np.where`

You can combine pandas with NumPy:

```
import numpy as np

df["result"] = np.where(
    df["score"] >= 60,
    "Pass",
    "Fail"
)
```

This is useful for simple conditional transformations.

---

24. Removing Columns

Remove one column:

```
df = df.drop(columns=["age"])
```

Multiple:

```
df = df.drop(
    columns=["age", "city"]
)
```

Remember that many pandas operations return a new object unless you explicitly choose an in-place or assignment-based approach.

---

25. Renaming Columns

```
df = df.rename(
    columns={
        "name": "student_name",
        "score": "exam_score"
    }
)
```

This is useful when incoming datasets have unclear or inconsistent column names.

---

26. Index Fundamentals

Every DataFrame has an index.

Example:

```
      name  score
0     Ali     85
1    Sara     92
2    Asim     78
```

Here:

```
0
1
2
```

are index labels.

You can create a custom index:

```
df = pd.DataFrame(
    {
        "score": [85, 92, 78]
    },
    index=["Ali", "Sara", "Asim"]
)
```

Now:

```
df.loc["Sara"]
```

selects Sara's row.

---

27. Resetting the Index

After filtering or other transformations, you may want a fresh sequential index.

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

Now the index becomes:

```
0
1
2
...
```

This is particularly useful after operations where the original index no longer represents something meaningful.

---

28. Reading CSV Files

One of pandas' most common uses is loading tabular data.

```
df = pd.read_csv("students.csv")
```

Then:

```
print(df.head())
```

The pandas getting-started documentation explicitly includes reading and writing tabular data as a core workflow.

---

29. Reading Excel Files

```
df = pd.read_excel("students.xlsx")
```

Specify a sheet:

```
df = pd.read_excel(
    "students.xlsx",
    sheet_name="Students"
)
```

pandas supports a broad range of data input and output operations, including CSV and Excel workflows.

---

30. Reading JSON

```
df = pd.read_json("students.json")
```

For JSON received from an API, you may also create a DataFrame from parsed Python data.

For example:

```
data = [
    {"name": "Ali", "score": 85},
    {"name": "Sara", "score": 92}
]

df = pd.DataFrame(data)
```

---

31. Exporting Data

CSV:

```
df.to_csv(
    "cleaned_students.csv",
    index=False
)
```

Excel:

```
df.to_excel(
    "cleaned_students.xlsx",
    index=False
)
```

The `index=False` prevents pandas from writing the DataFrame index as an extra column.

---

32. Missing Data

Real datasets are rarely perfect.

You may encounter:

`None`
`NaN`
`<NA>`
`NaT`

depending on the data type and context.

pandas provides dedicated missing-data tools such as:

`df.isna()`
`df.notna()`
`df.dropna()`
`df.fillna()`

The current pandas documentation describes several missing-value representations and provides dedicated methods for detecting, removing, and filling missing data.

---

33. Detecting Missing Values

`df.isna()`

Count missing values per column:

`df.isna().sum()`

Example:

```
name      0
age       2
score     1
city      0
```

This immediately tells you where the dataset needs attention.

---

34. Removing Missing Rows

```
cleaned = df.dropna()
```

This removes rows containing missing values.

You can target specific columns:

```
cleaned = df.dropna(
    subset=["score"]
)
```

This is useful when a particular field is essential.

---

35. Filling Missing Values

For example:

```
df["score"] = df["score"].fillna(0)
```

Or use a meaningful statistic:

```
df["score"] = df["score"].fillna(
    df["score"].mean()
)
```

The correct strategy depends on what the missing value means.

Never automatically replace missing values with zero without considering the data's meaning.

---

36. Detecting Duplicates

Check duplicate rows:

```
df.duplicated()
```

Count them:

```
df.duplicated().sum()
```

Remove them:

```
df = df.drop_duplicates()
```

For selected columns:

```
df = df.drop_duplicates(
    subset=["email"]
)
```

This is useful when each email should represent one unique record.

---

37. Data Types

Inspect:

```
df.dtypes
```

For example:

```
name     string
age       int64
score     float64
```

Convert a column:

```
df["age"] = df["age"].astype("int64")
```

For numeric conversion where invalid values may exist:

```
df["score"] = pd.to_numeric(
    df["score"],
    errors="coerce"
)
```

Invalid values become missing rather than crashing the entire conversion.

---

38. Working With Strings

pandas provides the `.str` accessor for vectorized string operations.

Convert to lowercase:

```
df["name"] = df["name"].str.lower()
```

Remove whitespace:

```
df["name"] = df["name"].str.strip()
```

Check whether a value contains text:

```
df[
    df["email"].str.contains(
        "@gmail.com",
        na=False
    )
]
```

Replace text:

```
df["city"] = df["city"].str.replace(
    "Lahore ",
    "Lahore",
    regex=False
)
```

This is extremely useful when cleaning text-heavy datasets.

---

39. Working With Dates

Convert a column to datetime:

```
df["date"] = pd.to_datetime(
    df["date"]
```

```
)
```

Then extract:

```
df["year"] = df["date"].dt.year
df["month"] = df["date"].dt.month
df["day"] = df["date"].dt.day
```

You can also filter:

```
df[
    df["date"] >= "2026-01-01"
]
```

Datetime handling is a major pandas use case, especially for time-series data.

---

40. Basic Statistics

Suppose:

```
scores = df["score"]
```

You can calculate:

```
scores.mean()
scores.median()
scores.min()
scores.max()
scores.sum()
scores.std()
scores.count()
```

For a DataFrame:

```
df.describe()
```

Many pandas statistical operations automatically exclude missing values where appropriate.

---

41. Counting Values

Suppose:

```
df["city"]
```

Use:

```
df["city"].value_counts()
```

Example:

```
Lahore      120
Islamabad    85
Karachi      70
Gujranwala   45
```

This is useful for categorical data.

---

42. GroupBy

`groupby()` is one of pandas' most important concepts.

Suppose:

```
df = pd.DataFrame({
    "city": [
        "Lahore",
        "Lahore",
        "Islamabad",
        "Islamabad"
    ],
    "sales": [100, 150, 200, 120]
})
```

Calculate sales per city:

```
df.groupby("city")["sales"].sum()
```

Conceptually:

```
Split
  
Group by city
  
Calculate sum
  
Combine results
```

pandas describes grouping as a **split-apply-combine** process.

---

43. Multiple Aggregations

```
summary = df.groupby("city")["sales"].agg(
    ["sum", "mean", "count"]
```

```
)
```

You can calculate several metrics at once.

Example:

```
           sum    mean    count
Islamabad   320    160       2
Lahore      250    125       2
```

---

44. Grouping Multiple Columns

```
df.groupby(
   ["city", "category"]
)["sales"].sum()
```

This creates groups based on combinations.

For example:

```
city       category
Lahore     Software
Lahore     Hardware
Islamabad  Software
Islamabad  Hardware
```

This is useful for multi-dimensional analysis.

---

45. `agg()` for Custom Summaries

You can define different calculations for different columns:

```
summary = df.groupby("city").agg(
    total_sales=("sales", "sum"),
    average_sales=("sales", "mean"),
    transactions=("sales", "count")
)
```

This produces a cleaner analytical table.

---

46. Applying Functions

For custom transformations, pandas provides tools such as `map`, `apply`, and related operations.

For simple element-wise transformation:

```
df["score"] = df["score"].map(
    lambda x: x + 5
)
```

However, prefer built-in vectorized operations when they can express the same transformation.

For example, this:

```
df["score"] = df["score"] + 5
```

is clearer than:

```
df["score"] = df["score"].map(
    lambda x: x + 5
)
```

---

47. Combining DataFrames With `concat`

Suppose:

```
january = pd.DataFrame({
    "name": ["Ali", "Sara"],
    "sales": [100, 150]
})

february = pd.DataFrame({
    "name": ["Asim", "Hina"],
    "sales": [120, 180]
})
```

Combine them:

```
all_sales = pd.concat(
    [january, february],
    ignore_index=True
)
```

Result:

```
   name  sales
0   Ali    100
1  Sara    150
2  Asim    120
3  Hina    180
```

`concat()` is useful when datasets have the same general structure and need to be combined along an axis.

---

48. Combining DataFrames With `merge`

Suppose one DataFrame contains customers:

```
customers = pd.DataFrame({
    "customer_id": [1, 2, 3],
    "name": ["Ali", "Sara", "Asim"]
})
```

Another contains orders:

```
orders = pd.DataFrame({
    "customer_id": [1, 2, 1],
    "amount": [100, 200, 150]
})
```

Merge them:

```
result = pd.merge(
    customers,
    orders,
    on="customer_id"
)
```

Now order records contain customer information.

This is similar to SQL joins.

pandas' `merge()` provides SQL-style join operations.

---

49. Join Types

Common merge types include:

```
inner
left
right
outer
```

Example:

```
pd.merge(
    customers,
    orders,
    on="customer_id",
    how="left"
)
```

A left join keeps every row from the left DataFrame and matches data from the right where available.

Choosing the correct join type is critical when combining datasets.

---

50. concat vs merge

Use `concat()` when you are generally stacking or combining pandas objects along an axis.

```
January
February
March
  □
concat
```

□
All months

Use merge() when you need to match records based on keys.

Customers
+
Orders
□
customer_id
□
merge

Mental model:

concat □ stack
merge □ match

---

51. Cleaning a Real Dataset

A common pandas pipeline looks like:

```
df = pd.read_csv("customers.csv")

df = df.drop_duplicates()

df["name"] = df["name"].str.strip()

df["email"] = df["email"].str.lower()

df["age"] = pd.to_numeric(
    df["age"],
```

```
    errors="coerce"
)

df["signup_date"] = pd.to_datetime(
    df["signup_date"],
    errors="coerce"
)

df = df.dropna(
    subset=["email"]
)
```

This is a realistic example of transforming messy input into cleaner data.

---

52. The Data Cleaning Mental Model

Think:

```
Raw Data
   
Inspect
   
Understand columns
   
Check types
   
Find missing values
   
Find duplicates
   
Normalize text
   
```

Convert types
  ↓
Validate
  ↓
Clean Data

Do not start changing data before understanding what is actually wrong.

---

53. Avoid Accidental Data Changes

Keep raw data separate from cleaned data.

For example:

```
raw_df = pd.read_csv("customers.csv")

clean_df = raw_df.copy()
```

Then transform:

```
clean_df["email"] = (
    clean_df["email"]
    .str.strip()
    .str.lower()
)
```

This makes debugging easier because you still have the original DataFrame.

---

54. Validate Your Cleaning

After cleaning, ask:

```
print(clean_df.shape)
print(clean_df.isna().sum())
print(clean_df.duplicated().sum())
print(clean_df.dtypes)
```

You can also validate business rules:

```
assert clean_df["age"].ge(0).all()
assert clean_df["age"].le(120).all()
```

Validation is especially important before exporting cleaned data or using it in another system.

---

55. A Complete Practical Example

Imagine a sales dataset:

```
import pandas as pd

df = pd.DataFrame({
    "customer": [
        "Ali",
        "Sara",
        "Asim",
        "Ali",
        "Hina"
    ],
    "city": [
        "Lahore",
        "Islamabad",
```

```
        "Lahore",
        "Lahore",
        "Islamabad"
    ],
    "sales": [
        100,
        250,
        150,
        200,
        300
    ]
})
```

Inspect:

```
print(df.head())
print(df.info())
```

Filter large sales:

```
large_sales = df[
    df["sales"] >= 200
]
```

Calculate total:

```
total_sales = df["sales"].sum()
```

Calculate city totals:

```
city_sales = (
    df.groupby("city")["sales"]
    .sum()
)
```

Sort:

```
city_sales = city_sales.sort_values(
    ascending=False
)
```

Create a report:

```
report = pd.DataFrame({
    "city": city_sales.index,
    "total_sales": city_sales.values
})
```

Export:

```
report.to_csv(
    "sales_report.csv",
    index=False
)
```

This is a complete mini data-processing pipeline.

---

56. Pandas + NumPy

pandas and NumPy are closely related.

NumPy:

```
Numerical arrays
```

pandas:

Labeled tabular data

For example:

```
import numpy as np
import pandas as pd

scores = np.array([
    80, 90, 75, 88
])

df = pd.DataFrame({
    "score": scores
})
```

You can use NumPy operations inside pandas workflows.

CSV

pandas

DataFrame

NumPy-style numerical operations

Analysis

pandas is built on top of NumPy and is intended to integrate with the broader scientific Python ecosystem.

---

57. Pandas vs NumPy

Feature	NumPy	pandas
Main structure	`ndarray`	`Series`, `DataFrame`
Labels	Limited	Built in
Tabular data	Possible	Excellent
Numerical arrays	Excellent	Good
Missing data tools	More basic	Extensive
Grouping	Limited	Powerful
CSV workflows	Not the main focus	Excellent
SQL-style joins	Not the main focus	Yes
Data cleaning	Basic building blocks	Extensive
Data analysis	Numerical foundation	High-level tabular analysis

A common relationship is:

`NumPy`

numerical foundation

pandas

labeled data manipulation

---

58. Performance and Memory

pandas is powerful, but not every dataset fits comfortably in memory.

For larger datasets, consider:

Select only required columns
Use efficient dtypes
Process data in chunks
Avoid unnecessary copies
Filter early
Use databases or specialized tools when appropriate

The pandas documentation has a dedicated section on scaling to larger datasets, including loading less data, using efficient data types, chunking, and considering other libraries when appropriate.

---

59. Common pandas Mistakes

Mistake 1: Not inspecting the data

Do not immediately write transformations.

Start with:

```
df.head()
df.shape
df.info()
df.dtypes
```

Mistake 2: Using the wrong selection method

Remember:

```
loc  → labels
iloc → positions
```

Mistake 3: Ignoring missing data

Always inspect:

```
df.isna().sum()
```

Mistake 4: Accidentally modifying raw data

Keep a clean copy:

```
clean_df = raw_df.copy()
```

Mistake 5: Using loops unnecessarily

Prefer pandas vectorized operations where possible.

Mistake 6: Confusing `concat` and `merge`

`concat` → combine along an axis
`merge` → match using keys

Mistake 7: Filling every missing value with zero

Zero may have a real meaning and may not represent “unknown.”

Mistake 8: Ignoring data types

A column containing numbers may actually be stored as strings.

Check:

```
df.dtypes
```

Mistake 9: Exporting the index accidentally

For normal CSV output:

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

---

60. Best Practices

1. Inspect first

```
df.head()
df.shape
df.info()
df.describe()
```

2. Keep raw and cleaned data separate

```
clean_df = raw_df.copy()
```

3. Use meaningful column names

Prefer:

```
customer_id
order_date
total_amount
```

over:

```
x
data1
value
```

4. Make transformations explicit

```
df["email"] = (
    df["email"]
    .str.strip()
    .str.lower()
)
```

5. Validate important assumptions

```
assert df["customer_id"].notna().all()
```

6. Prefer vectorized operations

Use pandas operations rather than unnecessary Python loops.

7. Understand your joins

Before merging, know:

```
What is the key?
Is it unique?
What happens when there is no match?
```

8. Export deliberately

Specify:

```
index=False
```

when the index is not part of the dataset.

---

61. Mini Project: Sales Data Analyzer

Create a file called:

```
sales.csv
```

with:

```
customer,city,category,amount
Ali,Lahore,Software,500
Sara,Islamabad,Hardware,300
Asim,Lahore,Software,700
Hina,Islamabad,Software,450
Ali,Lahore,Hardware,250
```

Your program should:

1. Load the CSV.
2. Inspect the dataset.
3. Check for missing values.
4. Remove duplicates.
5. Calculate total sales.
6. Calculate average sale.
7. Find the highest sale.
8. Calculate sales by city.
9. Calculate sales by category.
10. Find customers with sales above 400.
11. Sort sales from highest to lowest.
12. Export a summary CSV.

---

62. Mini Project Solution

```
import pandas as pd

df = pd.read_csv("sales.csv")

print("Shape:", df.shape)
print(df.head())
```

```
print("\nMissing values:")
print(df.isna().sum())

df = df.drop_duplicates()

total_sales = df["amount"].sum()
average_sale = df["amount"].mean()
highest_sale = df["amount"].max()

city_summary = (
    df.groupby("city")["amount"]
    .sum()
    .sort_values(ascending=False)
)

category_summary = (
    df.groupby("category")["amount"]
    .sum()
    .sort_values(ascending=False)
)

high_value = df[
    df["amount"] > 400
].sort_values(
    "amount",
    ascending=False
)

summary = pd.DataFrame({
    "metric": [
        "total_sales",
        "average_sale",
        "highest_sale"
    ],
```

```
    "value": [
        total_sales,
        average_sale,
        highest_sale
    ]
})

summary.to_csv(
    "sales_summary.csv",
    index=False
)

print("\nCity Summary:")
print(city_summary)

print("\nCategory Summary:")
print(category_summary)

print("\nHigh Value Sales:")
print(high_value)
```

This demonstrates a realistic pandas workflow:

Read

Inspect

Clean

Filter

Group

Sort

Summarize
Export

---

63. 5-Minute Challenge

Create this DataFrame:

```
df = pd.DataFrame({
    "name": [
        "Ali",
        "Sara",
        "Asim",
        "Hina",
        "Zara"
    ],
    "score": [
        72,
        91,
        64,
        88,
        95
    ],
    "city": [
        "Lahore",
        "Islamabad",
        "Lahore",
        "Karachi",
        "Islamabad"
    ]
})
```

Without writing a loop:

1. Find students scoring above 80.
2. Calculate average score.
3. Find the highest score.
4. Count students per city.
5. Sort students by score descending.
6. Add a `passed` column.
7. Create a city-wise average score table.

---

64. Exercises

Exercise 1

Create a DataFrame from a dictionary.

Exercise 2

Select one column as a Series.

Exercise 3

Select three columns as a DataFrame.

Exercise 4

Use `loc` to select a specific row.

Exercise 5

Use `iloc` to select the first three rows.

Exercise 6

Filter records using two conditions.

Exercise 7

Sort a DataFrame by two columns.

Exercise 8

Create a calculated column.

Exercise 9

Find missing values in every column.

Exercise 10

Fill missing numerical values with the column mean.

Exercise 11

Remove duplicate records.

Exercise 12

Convert a text column to lowercase.

Exercise 13

Convert a date column with `pd.to_datetime()`.

Exercise 14

Calculate grouped totals with `groupby()`.

Exercise 15

Combine two DataFrames with `concat()`.

Exercise 16

Combine two DataFrames using `merge()`.

Exercise 17

Export the cleaned DataFrame to CSV.

Exercise 18

Build a complete cleaning pipeline from a raw CSV.

---

65. Mini Quiz

1. What is a pandas DataFrame?

A. A Python function
B. A two-dimensional labeled tabular data structure
C. A database server
D. A NumPy scalar

Answer: B

2. What is a Series?

A. A one-dimensional labeled data structure
B. A two-dimensional database
C. A file format
D. A Python module

Answer: A

3. What does `df.shape` return?

A. Data types
B. Column names
C. Number of rows and columns
D. Missing values

Answer: C

4. What is the difference between `loc` and `iloc`?

A. `loc` uses labels; `iloc` uses integer positions
B. `loc` uses integers; `iloc` uses labels
C. They are identical
D. `iloc` only works with strings

Answer: A

5. Which method detects missing values?

A. `missing()`
B. `isna()`
C. `empty()`
D. `null()`

Answer: B

6. What does `groupby()` enable?

A. Splitting data into groups and applying calculations
B. Installing pandas
C. Creating Python classes
D. Compressing files

Answer: A

7. What is `merge()` commonly used for?

A. Sorting data
B. Joining DataFrames using related keys
C. Removing duplicates
D. Creating charts

Answer: B

8. What does `concat()` generally do?

A. Combines pandas objects along an axis
B. Converts strings to integers
C. Removes missing data
D. Creates indexes

Answer: A

---

66. Interview Questions

Beginner

1. What is pandas?
2. What is a Series?
3. What is a DataFrame?
4. What is an index?
5. What does `df.shape` return?
6. What does `df.info()` tell you?
7. What is the purpose of `df.describe()`?
8. How do you select a column?

Intermediate

9. What is the difference between `loc` and `iloc`?
10. How do you filter rows?
11. How do you handle missing values?
12. How do you remove duplicates?
13. How do you change a column's data type?
14. What is `groupby()`?
15. What is the difference between `merge()` and `concat()`?
16. How do you sort a DataFrame?
17. How do you create a calculated column?
18. How do you work with dates in pandas?

Advanced

19. Explain the split-apply-combine pattern.
20. What happens when you perform a merge with duplicate keys?
21. How would you validate a data-cleaning pipeline?
22. How would you process a dataset that does not fit comfortably in memory?

23. Why should unnecessary copies be avoided?
24. When would you use `apply()` versus a vectorized operation?
25. How would you design a reproducible pandas data pipeline?
26. How would you combine pandas with NumPy?
27. What problems can incorrect data types cause?
28. How would you debug a dataset after a merge unexpectedly increases its row count?

---

67. Pandas Cheat Sheet

Import

```
import pandas as pd
```

Series

```
pd.Series([1, 2, 3])
```

DataFrame

```
pd.DataFrame({
   "name": ["Ali", "Sara"],
   "score": [85, 92]
})
```

Read CSV

```
pd.read_csv("data.csv")
```

Read Excel

```
pd.read_excel("data.xlsx")
```

Write CSV

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

First rows

```
df.head()
```

Last rows

```
df.tail()
```

Shape

```
df.shape
```

Columns

```
df.columns
```

Data types

```
df.dtypes
```

Overview

```
df.info()
```

Statistics

```
df.describe()
```

Select column

```
df["score"]
```

Select multiple columns

```
df[["name", "score"]]
```

Label selection

```
df.loc[0, "score"]
```

Position selection

```
df.iloc[0, 1]
```

Filter

```
df[df["score"] > 80]
```

Sort

```
df.sort_values(
    "score",
    ascending=False
)
```

Missing values

```
df.isna()
df.isna().sum()
```

Drop missing values

```
df.dropna()
```

Fill missing values

```
df.fillna(0)
```

Duplicates

```
df.duplicated()
df.drop_duplicates()
```

Group

```
df.groupby("city")["sales"].sum()
```

Combine

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

Merge

```
pd.merge(
    df1,
    df2,
    on="id"
)
```

Rename

```
df.rename(
    columns={"old": "new"}
)
```

Drop column

```
df.drop(
    columns=["age"]
)
```

Datetime

```
pd.to_datetime(df["date"])
```

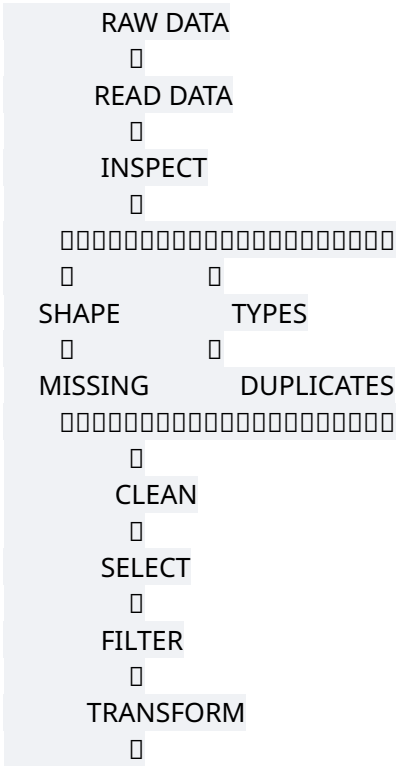
Numeric conversion

```
pd.to_numeric(
    df["score"],
    errors="coerce"
)
```

---

68. The Pandas Mental Model

When working with a new dataset, think:



```
GROUP
  ↓
COMBINE
  ↓
ANALYZE
  ↓
EXPORT
```

The goal is not to memorize hundreds of pandas methods.

The goal is to understand the **data workflow** and know which pandas operation represents each step.

---

69. Key Takeaways

1. pandas is a Python library for structured data manipulation and analysis.
2. Its two fundamental structures are `Series` and `DataFrame`.
3. A DataFrame represents labeled tabular data.
4. Always inspect a dataset before transforming it.
5. `loc` selects by labels; `iloc` selects by integer positions.
6. Boolean filtering is one of the most important pandas techniques.
7. pandas provides dedicated tools for missing data and duplicate records.
8. Columns can be transformed and new columns can be calculated without explicit Python loops.
9. `groupby()` implements the split-apply-combine pattern.
10. `concat()` generally combines objects along an axis.
11. `merge()` joins related datasets using keys.
12. pandas handles CSV, Excel, JSON, and many other data sources.
13. Dates and text can be transformed with pandas-specific accessors and functions.
14. pandas and NumPy complement each other.
15. Good pandas work is not just about manipulating data; it is about building a **clear, validated, reproducible data pipeline**.

References

pandas Documentation: https://pandas.pydata.org/docs/

10 Minutes to pandas: https://pandas.pydata.org/docs/user_guide/10min.html

pandas User Guide: https://pandas.pydata.org/docs/user_guide/

pandas Getting Started Tutorials: https://pandas.pydata.org/docs/getting_started/intro_tutorials/

Missing Data Guide: https://pandas.pydata.org/docs/user_guide/missing_data.html

Visit haas.dev for more resources and guides.

Website: https://dev-roast-app.vercel.app

Resources: https://dev-roast-app.vercel.app/resources