pandas Python data frames

START HERE

`import pandas as pd`
The usual import

`df.assign(<new>=...)`
Add a column

`pd.read_csv("<file>")`
Load a table

`df.groupby("<key>").sum()`
Summarise per group

`df.info()`
Columns, types, nulls

`df.merge(<other>, on="<key>")`
Join two tables

`df[df["<col>"] > 0]`
Filter rows

`df.to_parquet("<file>")`
Save the result

Read read_csv, read_parquet → **inspect** → **Clean** select, astype, fillna → **transform** → **Shape** groupby, merge, pivot → **export** → **Write** to_parquet, to_csv

CONCEPTS

A DataFrame is a set of Series (columns) sharing one index (row labels). Most operations align on that index, not on position.

Copy-on-Write
pandas 3 uses Copy-on-Write: any derived frame or column acts as a copy. Chained assignment like df["a"][0] = 1 never changes df; use df.loc[0, "a"] = 1.

READ AND WRITE

`pd.read_csv("<file>")`
Read a CSV file
Also takes URLs and .gz/.zip files.

`pd.read_csv("<file>", parse_dates=["<col>"])`
Parse a column as dates

`pd.read_parquet("<file>")`
Read a Parquet file
Needs pyarrow installed.

`pd.read_excel("<file>", sheet_name="<name>")`
Read an Excel sheet
Needs openpyxl for .xlsx.

`pd.DataFrame({"a": [1, 2], "b": ["x", "y"]})`
Build a frame from a dict

`df.to_csv("<file>", index=False)`
Write CSV without the index
Without index=False the row labels become an unnamed column.

`df.to_parquet("<file>")`
Write Parquet (keeps dtypes)

INSPECT

`df.head(<n>) / df.tail(<n>)`
First or last rows (default 5)

`df.info()`
Columns, dtypes, non-null counts

`df.describe()`
Summary statistics
Numeric and date columns only; include="all" adds the rest.

`df.shape`
(rows, columns)

`df.dtypes`
Type of each column

`df["<col>"].value_counts()`
Count each distinct value
Add normalize=True for shares, dropna=False to count NaN.

`df["<col>"].unique()`
Distinct values

SELECT

`df["<col>"]`
One column as a Series

`df[["<a>", "<b>"]]`
Several columns as a DataFrame

`df.loc[<rows>, <cols>]`
Select by label
Label slices include the end: df.loc["a":"c"] contains "c".

`df.iloc[<rows>, <cols>]`
Select by position
Position slices exclude the end, like Python lists.

`df[df["<col>"] > 0]`
Filter rows with a mask

`df[(<cond1>) & (<cond2>)]`
Combine conditions
Use & | ~ with brackets around each test, not and/or/not.

`df[df["<col>"].isin([...])]`
Rows whose value is in a list

`df.query("<col> > 0 and <key> == 'x'")`
Filter with an expression string
Refer to Python variables as @name.

CHANGE COLUMNS

`df["<new>"] = df["<a>"] * 2`
Add or overwrite a column

`df.assign(<new>=lambda d: d.<a> + d.<b>)`
Add columns, return a new frame
The lambda sees the frame at that point of a method chain.

`df.loc[<mask>, "<col>"] = <value>`
Set values where a condition holds

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

`df.drop(columns=["<col>"])`
Remove columns

`df["<col>"].astype("<dtype>")`
Convert type (int64, float, str, category)

`pd.to_numeric(df["<col>"], errors="coerce")`
Text to numbers, bad values to NaN

`df["<col>"].map({<k>: <v>})`
Translate values via a dict
Values missing from the dict become NaN.

MISSING DATA

`df.isna().sum()`
Count missing values per column

`df.dropna()`
Drop rows with any missing value
subset=["<col>"] checks only those columns.

`df.fillna(<value>)`
Fill missing values
A dict fills per column: fillna({"a": 0}).

SORT AND DEDUPE

`df.sort_values("<col>", ascending=False)`
Sort rows by a column

`df.drop_duplicates(subset=["<col>"])`
Drop duplicate rows
keep="last" keeps the last copy instead of the first.

`df.set_index("<col>")`
Use a column as row labels

`df.reset_index(drop=True)`
Renumber rows 0..n-1
Without drop=True the old index becomes a column.

GROUP AND AGGREGATE

`df.groupby("<key>")["<col>"].sum()`
Total per group

`df.groupby("<key>")["<col>"].agg(["mean", "max"])`
Several stats at once

`df.groupby("<key>").agg(<out>=("<col>", "sum"))`
Named aggregation
Gives flat, named result columns.

`df.groupby("<key>")["<col>"].transform("mean")`
Group stat on every row
Result has the original length, so it can be a new column.

`df["<col>"].mean()`
Column mean
(also .sum, .max, .count)

COMBINE

`df.merge(<other>, on="<key>")`
Join on a column (inner)

`df.merge(<other>, on="<key>", how="left")`
Left join
how is inner, left, right, outer or cross.

`pd.concat([<df1>, <df2>], ignore_index=True)`
Stack frames vertically

RESHAPE

`df.pivot_table(index="<r>", columns="<c>", values="<v>", aggfunc="sum")`
Spreadsheet-style pivot
Default aggfunc is mean; fill_value=0 replaces gaps.

`df.melt(id_vars=["<id>"], var_name="<k>", value_name="<v>")`
Wide to long

TEXT

`df["<col>"].str.lower()`
Lowercase (any str method)

`df["<col>"].str.contains("<pat>", na=False)`
Regex match as a mask
Pass regex=False for a literal substring.

DATES AND TIME SERIES

`pd.to_datetime(df["<col>"])`
Parse text as datetimes
format="%d.%m.%Y" for unambiguous, faster parsing.

`df["<col>"].dt.year`
Date part
(also .month, .day_name())

`df.resample("ME", on="<date>").sum()`
Aggregate per calendar month
Use "D", "W", "ME" (month end), "QE", "YE"; plain "M" is no longer accepted.

APPLY VS VECTORISED

`df["<a>"] + df["<b>"]`
Column arithmetic, no loops

`df["<col>"].apply(<func>)`
Call a function per value
A Python-level loop: slow on big data.

`df.apply(<func>, axis=1)`
Call a function per row
Slowest option; try a vectorised expression first.

RECIPES

Load, clean, save

```
df = pd.read_csv("sales.csv")
df = (df
  .rename(columns=str.lower)
  .dropna(subset=["amount"])
  .drop_duplicates())
df.to_parquet("sales.parquet")
```

Report per group

```
top = (df.groupby("region")
  .agg(total=("amount", "sum"),
       orders=("id", "count"))
  .nlargest(10, "total"))
print(top)
```