Pandas con esempi

1) `pd.read_csv` / `pd.read_excel`

Due funzioni per **importare dati tabellari** in un DataFrame. `read_csv` legge file testuali separati (comma, punto e virgola, ecc.); `read_excel` importa da fogli Excel (richiede un engine tipo `openpyxl`).

È buona pratica specificare i **tipi** e convertire date già in lettura (`parse_dates`), così eviti sorprese nei passaggi successivi.

```
import pandas as pd

# CSV con separatore ';' e parse date
ordini = pd.read_csv("ordini.csv", sep=";", parse_dates=["order_date"])

# Excel: foglio specifico
clienti = pd.read_excel("clienti.xlsx", sheet_name="anagrafica")
```

Punti chiave

- `parse_dates=["col"]` per colonne data.
- `usecols`, `dtype` per performance e coerenza.
- Separa **I/O** dalla logica: carica prima, elabora dopo.

---

2) `pd.merge` (join stile SQL)

Unisce due DataFrame su una o più **chiavi**. `how` controlla il tipo di join: `inner`, `left`, `right`, `outer`.

Per evitare colonne duplicate, controlla i nomi e usa `suffixes=('_a','_b')` quando servono.

```
import pandas as pd

prodotti = pd.DataFrame({"sku":[1,2,3], "prezzo":[10,20,30]})
stock    = pd.DataFrame({"sku":[2,3,4], "qta":[50,0,7]})
```

```
# Mantieni tutti i prodotti, aggiungi qta dove disponibile
df = pd.merge(prodotti, stock, on="sku", how="left")
```

Punti chiave

- `on=` chiave/e; se i nomi differiscono: `left_on`, `right_on`.
- `how='left'` conserva la cardinalità della tabella di sinistra.
- Usa `join` sugli **index** quando le chiavi sono negli indici.

---

3) Maschere booleane e `.loc` / `.iloc`

I filtri si fanno con **serie booleane**; `.loc` filtra per **etichette**, `.iloc` per **posizione**.

Combina condizioni con `&` e `|` (non usare `and` / `or` su Series).

```
import pandas as pd

df = pd.DataFrame({"città":["Roma","Milano","Roma"], "vendite":[100,250,80]})
roma_alta = df.loc[(df["città"]=="Roma") & (df["vendite"]>90), ["città","vendite"]]
```

Punti chiave

- `df[condizione]` ≈ `df.loc[condizione]`.
- `&`, `|`, `~` per AND/OR/NOT bitwise.
- `.iloc[r1:r2, c1:c2]` per slicing posizionale.

---

4) Creare colonne derivate e operazioni vettoriali

Le colonne sono **Series**: somma, moltiplica, concatena in modo **vectorized** (niente loop).

`assign` aiuta a creare più colonne in chain.

```
import pandas as pd

df = pd.DataFrame({"prezzo":[10,20,30], "qta":[2,0,5]})
```

```
df = df.assign(valore=lambda d: d["prezzo"]*d["qta"],
               is_zero=lambda d: d["qta"].eq(0))
```

Punti chiave

- Operazioni su colonne sono veloci e leggibili.
- `where`, `clip`, `round` utili per regole semplici.
- `assign` mantiene uno stile “a pipeline”.

---

5) `fillna`, `dropna`, `isna`

Gestione **valori mancanti (NaN)**: riempi (imputazione) o rimuovi righe/colonne.

Scegli la strategia in base alla semantica del dato (0? media? mediana?).

```
import pandas as pd
import numpy as np

df = pd.DataFrame({"a":[1, np.nan, 3], "b":[np.nan, 2, 2]})
df["a"] = df["a"].fillna(df["a"].mean())
df = df.dropna(subset=["b"])  # rimuovi righe dove b è NaN
```

Punti chiave

- `isna()` / `notna()` per diagnosticare.
- `fillna` accetta **scalari**, **Series** o metodi (`ffill`, `bfill`).
- Non “falsare” le analisi: documenta le imputazioni.

---

6) `drop_duplicates`

Evita duplicati secondo un **subset** di colonne.

Utile dopo merge o import non puliti.

```
import pandas as pd

df = pd.DataFrame({"id":[1,1,2], "val":[10,10,20]})
df = df.drop_duplicates(subset=["id"], keep="first")
```

Punti chiave

- Scegli `keep='first'|'last'|False`.
- Attento ai duplicati **logici** (stesse info, timestamp diverso).

---

7) Tipi e `to_datetime` / `astype`

Converti esplicitamente i **dtypes** per coerenza e performance.

Le date vanno convertite con `to_datetime`.

```
import pandas as pd

df = pd.DataFrame({"data":["2024-01-01","2024-03-05"], "codice":["00
1","002"]})
df["data"] = pd.to_datetime(df["data"])
df["codice"] = df["codice"].astype("string")
```

Punti chiave

- `astype` per numeri, categorie, stringhe.
- `to_numeric(..., errors="coerce")` per ripulire numeri sporchi.
- Date coerenti ⇒ facile fare `dt.year`, `dt.month`.

---

8) `groupby` + `agg` (named aggregations)

Raggruppa per una o più **chiavi** e calcola statistiche in un colpo solo.

Le **named aggregations** rendono colonne chiare e pulite.

```
import pandas as pd

df = pd.DataFrame({"cat":["A","A","B"], "val":[10,30,20]})
agg = df.groupby("cat").agg(
    n=("val","count"),
    media=("val","mean"),
    std=("val","std"),
)
```

Punti chiave

- Raggruppi multipli: `groupby(["a","b"])`.
- `.agg` accetta dict o named tuples.
- `as_index=False` se vuoi le chiavi come colonne.

---

9) `sort_values` / `sort_index`

Ordina per **colonne** (valori) o per **indice**.

Utile per ranking o per presentare tabelle leggibili.

```
import pandas as pd

df = pd.DataFrame({"prodotto":["x","y","z"], "ricavi":[200,50,120]})
top = df.sort_values("ricavi", ascending=False).head(2)
```

Punti chiave

- `ascending=[True, False]` per sort multi-colonna.
- `na_position='first'|'last'` per gestire NaN.
- Ordina prima di esportare o mostrare.

---

10) `quantile` + `transform` (IQR per outlier)

Per outlier stile **Tukey**, servono Q1/Q3 e IQR.

Con `transform` porti le statistiche **per gruppo** alla granulità riga.

```
import pandas as pd

df = pd.DataFrame({"reparto":["X","X","X","Y","Y"],
                   "score":[10,12,200,5,6]})

g = df.groupby("reparto")["score"]
q1 = g.transform(lambda s: s.quantile(0.25))
q3 = g.transform(lambda s: s.quantile(0.75))
iqr = q3 - q1
low, up = q1 - 1.5*iqr, q3 + 1.5*iqr
```

```
df["is_out"] = (df["score"]<low) | (df["score"]>up)
```

Punti chiave

- `transform` mantiene la **stessa shape** del DF originale.
- Outlier = fuori dai "fences" (Q1−1.5*IQR, Q3+1.5*IQR).
- Fai outlier **per reparto** (non globalmente).

---

11) Normalizzazione min–max **per gruppo**

Per confronti interni a un gruppo, normalizza 0–1 per **ogni gruppo**.

Serve per comporre punteggi con scale diverse.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({"team":["A","A","B","B"], "score":[50,100,10,20]})
grp = df.groupby("team")["score"]
min_ = grp.transform("min")
max_ = grp.transform("max")
norm = (df["score"] - min_) / (max_ - min_).replace(0, np.nan)
df["score_norm"] = norm.fillna(0)
```

Punti chiave

- Per pesare dimensioni diverse (es. rating + goal).
- Gestisci il caso `max==min` (divisione per zero).
- Normalizza **prima** di combinare punteggi.

---

12) Ranking con `sort_values` , `groupby().head()`

Ordina per punteggio e prendi i **Top N** globali o per gruppo.

Pattern semplice e leggibile.

```
import pandas as pd
```

```
df = pd.DataFrame({"team":["A","A","B","B"], "score":[0.3,0.9,0.2,0.6]})

top_global = df.sort_values("score", ascending=False).head(3)

top_per_team = (
    df.sort_values(["team","score"], ascending=[True, False])
      .groupby("team", group_keys=True)
      .head(1)
)
```

Punti chiave

- `groupby().head(n)` dopo il sort per top-k per gruppo.
- `nlargest(n, "col")` è un'alternativa veloce (solo una colonna).
- Conserva le chiavi nel sort per output leggibile.

---

13) `pd.ExcelWriter` + `to_excel` (multi-sheet)

Esporta più DataFrame in **fogli** diversi nello stesso file Excel.

Ottimo per un **cruscotto** compatto.

```
import pandas as pd

kpi = pd.DataFrame({"metrica":["A","B"], "val":[10,20]})
tab = pd.DataFrame({"x":[1,2], "y":[3,4]})

with pd.ExcelWriter("report.xlsx") as w:
    kpi.to_excel(w, sheet_name="KPI", index=False)
    tab.to_excel(w, sheet_name="Tabelle", index=False)
```

Punti chiave

- Crea `output/` se non esiste.
- Nomi **sheet** coerenti (KPI, Aggregati, ...).
- Evita fogli "enormi": meglio 3–4 fogli chiari.

---

14) `to_csv` (export mirato)

Scrivi CSV per viste specifiche (es. solo outlier).

Occhio all'`index` se non ti serve.

```
import pandas as pd

df = pd.DataFrame({"id":[1,2,3], "flag":[False, True, False]})
df[df["flag"]].to_csv("solo_flag_true.csv", index=False)
```

Punti chiave

- Imposta `index=False` se l'indice non è informativo.
- Encoding: `encoding="utf-8"` per compatibilità ampia.
- Mantieni **naming** coerente (es. `hr_outliers.csv`).

---

15) Statistiche rapide: `describe`, `mean`, `std`, `count`

Per una fotografia immediata del dataset: **tendenza centrale** e **dispersione**.

Ideale come controllo pre-aggregazioni complesse.

```
import pandas as pd

df = pd.DataFrame({"val":[10,11,9,50]})
print(df["val"].describe())   # count, mean, std, min, quartili, max
```

Punti chiave

- Ignora automaticamente i NaN (in molte funzioni).
- Usa su subset mirati (per reparto/ruolo) quando serve.
- Non sostituisce le aggregazioni "per chiave".

---

16) `apply` vs `map` vs `transform` (quando servono)

- `map`: element-wise su Series, per mapping semplice (dizionari, funzioni unarie).
- `apply`: su Series o DataFrame; flessibile, ma spesso meno performante del vettoriale.

- `transform`: come `apply`, ma **ritorna una Series allineata** alla shape di input (perfetto per logiche per-gruppo).

```
import pandas as pd

s = pd.Series(["Roma","Milano"])
# map: sostituzioni semplici
mappa = {"Roma":"RM", "Milano":"MI"}
s2 = s.map(mappa)

df = pd.DataFrame({"grp":["A","A","B"], "x":[1,2,10]})
# transform: media per gruppo, allineata a ogni riga
df["x_mean_grp"] = df.groupby("grp")["x"].transform("mean")
```

Punti chiave

- Preferisci **vectorized** prima di `apply`.
- `transform` è la chiave per feature “per-gruppo”.
- `map` per lookup puliti (es. codici → descrizioni).
