Introduzione a NumPy e Pandas in Python

Ambiente virtuale e pip

Per gestire in modo sicuro le dipendenze di un progetto Python si crea un **ambiente virtuale**. Questo è un albero di directory isolato che contiene un'installazione specifica di Python e i pacchetti necessari, evitando conflitti di versione con altri progetti [1]. Ad esempio, creando un nuovo ambiente con il modulo `venv` (`python -m venv mia_env`) si ottiene una copia autonoma dell'interprete Python [2]. Dopo la creazione, è necessario **attivare** l'ambiente (su Windows eseguendo `mia_env\Scripts\activate`, su Unix `source mia_env/bin/activate`) in modo che i comandi successivi (come `python` o `pip`) agiscano all'interno di questo ambiente [3] [4]. Una volta attivo, si usano **pip** (il gestore pacchetti di Python) per installare librerie aggiuntive dal Python Package Index (PyPI) [4]. In Visual Studio Code si può aprire il terminale integrato e digitare questi comandi, garantendo che l'ambiente attivo venga riconosciuto dall'editor.

```
# Crea un nuovo ambiente virtuale Python
python -m venv mia_env

# Attiva l'ambiente virtuale
# Windows:
mia_env\Scripts\activate
# Linux/Mac:
source mia_env/bin/activate

# Installa librerie all'interno dell'ambiente
pip install pandas numpy
```

- L'ambiente virtuale *isola* le librerie del progetto, evitando conflitti di versioni.
- **pip** installa pacchetti dal repository PyPI nell'ambiente attivo [4].
- Utilizzare un ambiente per ciascun progetto semplifica la **gestione delle dipendenze**.
- Ricordarsi di **attivare** l'ambiente (cambierà il prompt) prima di lanciare Python o Jupyter.

Jupyter Notebook

Un **notebook Jupyter** è un'applicazione web open-source per scrivere codice in modo interattivo [5]. Ogni notebook consiste in celle di **codice** (ad es. Python) e celle di testo (Markdown) che possono includere spiegazioni, immagini o equazioni. Questo formato è molto usato in data science perché permette di alternare codice eseguibile a risultati (grafici, tabelle) con spiegazioni narrative. I notebook sono ideali per l'analisi esplorativa dei dati (EDA), visualizzazione e documentazione dei passaggi, rendendo il lavoro trasparente e riproducibile [6] [7]. Ad esempio, il tutorial Databricks spiega che Jupyter permette di *"mostrare il lavoro"* combinando codice, annotazioni e immagini [7]. In Visual Studio Code è possibile aprire file `.ipynb` con estensione Jupyter, eseguendo celle direttamente nell'editor; alternativamente si avvia `jupyter notebook` dal terminale per lavorare nel browser.

```
# Esempio di cella in Jupyter
print("Benvenuti nel Jupyter Notebook!")
```

- Ambiente interattivo: codice e risultati appaiono cella per cella.
- Combinazione di **codice, annotazioni e grafici** facilita la documentazione del flusso di lavoro [5].
- I notebook sono eseguibili passo-passo ed esportabili (HTML, PDF, ecc.), utili per condividere analisi [7].
- Supportano pacchetti Python scientifici (NumPy, Pandas, Matplotlib) e kernel multipli.

NumPy: array e operazioni

NumPy è la libreria fondamentale per il calcolo numerico in Python [8]. **NumPy** fornisce array multidimensionali (ndarray) e operazioni vettoriali ad alte prestazioni. In pratica, sostituisce i loop Python lenti con operazioni interne ottimizzate su interi array. Il sito ufficiale descrive NumPy come una libreria *"open source...che aggiunge supporto a grandi matrici e array multidimensionali insieme a una vasta collezione di funzioni matematiche di alto livello"* [8]. Si creano array da liste Python (ad es. `np.array([1,2,3])`), impostando forma (*shape*) e tipo di dato. Questi array hanno attributi come `ndim`, `shape` e supportano l'indicizzazione/slicing simile alle liste Python. Le operazioni aritmetiche (somma, moltiplicazione, etc.) vengono applicate elemento-per-elemento in modo vettoriale.

```
import numpy as np

# Creazione di array 1D e 2D
arr1 = np.array([1, 2, 3])
arr2 = np.array([[1, 2, 3],
                 [4, 5, 6]])
print("Array 1D:", arr1)
print("Array 2D:\n", arr2)

# Operazioni aritmetiche vettoriali
somma = arr2 + 10  # Broadcasting: aggiunge 10 a tutti gli elementi
print("Somma (arr2 + 10):\n", somma)
```

- Gli array NumPy hanno forma fissa (*shape*) e possono contenere elementi numerici omogenei.
- Le operazioni come `arr + 10` o `arr1 * arr2` si applicano **elementwise** senza scrivere loop.
- **Broadcasting**: NumPy estende automaticamente array di forma minore per renderli compatibili nelle operazioni [9].
- Indici e slicing: si accede con `arr[i]`, `arr[i:j]`, o con slicing multi-dimensionale.

Serie e DataFrame

In Pandas le due strutture dati principali sono **Series** e **DataFrame** [10] [11]. Una *Series* è un array monodimensionale con un indice etichettato: si può pensare a una Series come a una colonna di dati (ad esempio una colonna di un foglio di calcolo). Un *DataFrame* è una tabella bidimensionale (righe e colonne), simile a un foglio di calcolo o a una tabella SQL: ha un indice di riga e nome di colonna etichettati, e ciascuna colonna può avere un tipo di dato diverso [12]. In pratica, un DataFrame è

composto da più Series combinate. Pandas è costruito su NumPy, rendendo queste strutture estremamente efficienti su dataset di grandi dimensioni [13]. Le Series e i DataFrame permettono operazioni di indicizzazione intelligenti: è possibile accedere alle righe/colonne tramite etichette (`.loc`) o posizioni intere (`.iloc`). Ad esempio, per un DataFrame `df`, `df['Nome']` restituisce la Series corrispondente alla colonna "Nome", mentre `df.loc[0]` o `df.iloc[0]` restituisce la prima riga [14].

```
import pandas as pd

# Creazione di una Series da lista (con indice personalizzato)
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
print("Series:")
print(s)

# Creazione di un DataFrame da dizionario
df = pd.DataFrame({
    'Nome': ['Alice', 'Bob', 'Charlie'],
    'Età': [25, 30, 22]
})
print("\nDataFrame:")
print(df)

# Indicizzazione e slicing
print("\nElemento della Series con indice 'b':", s['b'])
print("Prima riga del DataFrame con .loc:\n", df.loc[0])
print("Prime due righe con slicing:\n", df.iloc[0:2])
```

- **Creazione**: Series da lista, dizionario, array NumPy; DataFrame da dizionario di liste/array, da file CSV/Excel, ecc.
- **Indicizzazione**: accesso facilitato con etichette (`.loc`) o posizioni (`.iloc`) [14].
- È possibile selezionare colonne singole (`df['col']`), più colonne (`df[['col1','col2']]`) o righe intere.
- Slicing e filtraggio permettono di ottenere subset del DataFrame (ad es. `df[0:5]` per prime 5 righe, o con condizioni complesse).

Importazione ed esportazione di dati (CSV, Excel)

Pandas semplifica la lettura e la scrittura di dati strutturati. I file CSV (Comma-Separated Values) sono testi grezzi separati da virgole, largamente usati perché leggibili da qualsiasi applicazione [15]. Con `pd.read_csv('file.csv')` si carica un CSV in un DataFrame. Analogamente, `pd.read_excel('file.xlsx')` (richiede la libreria `openpyxl` o simili installata) importa dati da fogli Excel in un DataFrame [16]. Le funzioni `to_csv()` e `to_excel()` permettono l'esportazione dal DataFrame verso file su disco. Ad esempio, si può salvare un DataFrame come CSV: `df.to_csv('output.csv', index=False)`, e come Excel: `df.to_excel('output.xlsx', index=False)`. Queste funzioni supportano molte opzioni (delimitatori diversi, fogli multipli, encoding, ecc.). È importante prestare attenzione agli header e agli indici (i parametri `header`, `index`).

```python
import pandas as pd

# Leggi un CSV in un DataFrame
df_csv = pd.read_csv('dati.csv')  # file CSV di esempio
print("Prime righe di dati.csv:")
print(df_csv.head())

# Esporta DataFrame su CSV
df_csv.to_csv('dati_output.csv', index=False)

# Leggi un file Excel (foglio "Foglio1")
df_excel = pd.read_excel('dati.xlsx', sheet_name='Foglio1')
print("\nPrime righe di dati.xlsx:")
print(df_excel.head())

# Esporta DataFrame su Excel
df_excel.to_excel('dati_output.xlsx', index=False)
```

- `pd.read_csv()` e `pd.read_excel()` importano dati in DataFrame (rispettivamente CSV ed Excel) [15] [16].
- `df.to_csv()` e `df.to_excel()` esportano i dati su file.
- Possibilità di specificare fogli (`sheet_name`), separatori (`sep`), encoding (`encoding`), righe di header, colonne da leggere (`usecols`), ecc.
- I dati importati mantengono tipologie (numeri, stringhe, date) e indici che si possono personalizzare.

Filtri, selezioni e condizioni

Pandas consente di filtrare i dati di un DataFrame usando condizioni booleane su colonne. Ogni confronto (`>`, `<`, `==`, `!=`, ecc.) applicato a una Series genera una Series di valori booleani. Ad esempio, `df['Età'] > 18` restituisce una Series `True/False` per ogni riga. Si può quindi passare questa condizione per ottenere le righe che la soddisfano: `df[df['Età'] > 18]` ritorna un DataFrame con solo le righe dove l'età è maggiore di 18. In generale, l'espressione `df[condizione]` effettua un *filtro* sulle righe. Nel tutorial vediamo un esempio simile: `df[df['Salary'] > 60000]` seleziona dipendenti con salario alto [17]. Si possono combinare più condizioni usando operatori logici bitwise di NumPy: `&` (AND), `|` (OR), `~` (NOT). Per esempio, `df[(df['Età'] > 18) & (df['Genere'] == 'F')]` seleziona le donne maggiorenni.

```python
import pandas as pd

df = pd.DataFrame({
    'Nome': ['Anna', 'Luca', 'Maria', 'Giovanni'],
    'Età': [23, 17, 35, 29]
})

# Seleziona righe con condizione booleana (Età >= 18)
maggiorenni = df[df['Età'] >= 18]
print("Ragazzi maggiorenni:")
print(maggiorenni)
```

```
# Usa .loc con condizione e selezione di colonne
nomi_maggiorenni = df.loc[df['Età'] >= 18, ['Nome']]
print("\nNomi di età >= 18:")
print(nomi_maggiorenni)
```

- **Filtri booleani**: si usa `df[condizione]` dove `condizione` è un array di True/False sulla stessa lunghezza di `df`.
- `.loc` permette di combinare filtro righe e selezione colonne: `df.loc[condizione, ['col1','col2']]`.
- È possibile concatenare condizioni multiple con `&` (and) e `|` (or) invece di `and`, `or`.
- Pandas supporta anche il metodo `df.query("Età > 18 and Genere == 'F'")` per query in sintassi stringa.

Operazioni su colonne e righe

Un DataFrame Pandas consente di manipolare facilmente righe e colonne. Le **colonne** sono in sostanza Series: si possono creare nuove colonne semplicemente assegnando a `df['Nuova']` una Series o una lista. Ad esempio, `df['C'] = df['A'] + df['B']` aggiunge una colonna C ottenuta dalla somma di A e B. Per cancellare una colonna si usa `df.drop('Colonna', axis=1)`. Allo stesso modo, `df.drop(3, axis=0)` elimina la riga con indice 3. Inoltre, si possono rinominare etichette di colonne o righe con `df.rename({vecchio: nuovo}, axis=1)` o `axis=0` [18]. I DataFrame permettono operazioni vettoriali su intere colonne: per esempio `df['A'] * 2` raddoppia tutti i valori di A. Si possono applicare funzioni elementwise alle colonne (ad es. tramite `df['A'].map(lambda x: ...)` o `df.apply()`). Dal tutorial vediamo un caso in cui viene aggiunta la colonna "Salary" [17], illustrando quanto sia facile creare nuove colonne a partire dai dati esistenti.

```
import pandas as pd

df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [10, 20, 30]
})
print("DataFrame originale:")
print(df)

# Aggiungi una nuova colonna come combinazione di altre
df['C'] = df['A'] + df['B']  # somma colonna A e B
print("\nDopo aver aggiunto colonna C:")
print(df)

# Rimuovi la colonna B
df.drop('B', axis=1, inplace=True)
print("\nDopo aver rimosso la colonna B:")
print(df)

# Rinominare una colonna
df_renamed = df.rename(columns={'A': 'Alpha', 'C': 'Sum'})
```

```
print("\nDopo il rename delle colonne:")
print(df_renamed)
```

- Operazioni aritmetiche si propagano su intere colonne (es. `df['col'] + 5` aggiunge 5 a ogni riga).
- **Creare/Modificare colonne**: `df['Nuova'] = ...`; **Cancellare colonne**: `df.drop('col', axis=1)`.
- **Righe**: `df.drop(index, axis=0)` o uso di `.loc` / `.iloc` per selezionare/sovrascrivere righe.
- **Rinominare** colonne o indici con `df.rename()`, ad es. `df.rename(columns={'A':'a'})` [18].
- Si possono usare `df.apply()` o `df.applymap()` per applicare funzioni a righe/colonne o a tutti gli elementi.

Join e Merge fra DataFrame

Pandas supporta l'unione di DataFrame (join/merge) come farebbe un database SQL [19]. Il metodo principale è `pd.merge(df1, df2, on=chiave, how='inner')`, che combina due DataFrame sul campo comune `chiave`. Per default effettua un *inner join* (mantiene solo le chiavi comuni) [19] [20]. Ad esempio, unendo due DataFrame sulla colonna "id", i record con lo stesso `id` vengono fusi in una singola riga. Specificando `how='outer'` si ottiene una *union* completa (unisce tutte le chiavi, riempiendo con NaN quelle mancanti) [21]. Esistono anche join di tipo `left` e `right` per usare tutte le chiavi rispettivamente del DataFrame sinistro o destro [22] [23]. In alternativa, è possibile usare `df1.join(df2)` che unisce per indice. Per concatenare DataFrame in senso verticale (append di righe) o orizzontale (affiancamento di colonne) si usa `pd.concat([df1, df2], axis=0)` o `axis=1`. In sintesi, con `merge/join/concat` Pandas offre vari modi per unire e affiancare i dati.

```
import pandas as pd

df1 = pd.DataFrame({
    'id': [1, 2, 3],
    'Nome': ['A', 'B', 'C']
})
df2 = pd.DataFrame({
    'id': [2, 3, 4],
    'Valore': [100, 200, 300]
})

# Inner merge sui campi 'id' (soltanto chiavi comuni)
merged_inner = pd.merge(df1, df2, on='id', how='inner')
print("Inner merge su 'id':")
print(merged_inner)

# Outer merge (tutte le chiavi)
merged_outer = pd.merge(df1, df2, on='id', how='outer')
print("\nOuter merge su 'id' (tutte le chiavi):")
print(merged_outer)
```

- `pd.merge()` unisce come JOIN SQL: chiavi comuni e tipi (inner/outer/left/right) [19] [20].

- Con *inner join* (`how='inner'`) rimangono solo le chiavi presenti in entrambi i DF [20].
- Con *outer join* si mantengono tutte le chiavi, inserendo NaN dove mancano valori [21].
- `df1.join(df2)` unisce per indice di riga; `pd.concat()` concatena per righe o colonne.
- Utile per combinare dati da diverse sorgenti correlati da una colonna chiave (es. ID).

Raggruppamenti (groupby)

Il metodo `groupby()` consente di raggruppare le righe di un DataFrame in base ai valori di una (o più) colonna, per poi applicare funzioni di aggregazione [24]. Non si tratta di un semplice ordinamento, ma di *suddividere* il DataFrame in subset e poi *applicare* calcoli su ogni gruppo. Ad esempio, con `df.groupby('Team')['Score'].mean()` otteniamo la media dei punteggi per ogni squadra (Team). Il tutorial spiega che raggruppando su "Località" si ottengono aggregazioni separate per ciascuna città [24]. Si può aggregare con funzioni come `mean()`, `sum()`, `count()`, oppure passare dizionari a `.agg({'col': [funzioni]})` per più operazioni contemporaneamente. Il risultato è un oggetto GroupBy o DataFrame raggruppato a più livelli, a seconda del numero di chiavi di raggruppamento.

```
import pandas as pd

df = pd.DataFrame({
    'Team': ['A', 'A', 'B', 'B', 'A'],
    'Score': [10, 15, 10, 20, 25]
})

# Raggruppa per 'Team' e calcola la media dei punteggi
mean_score = df.groupby('Team')['Score'].mean()
print("Media dei punteggi per team:")
print(mean_score)

# Raggruppa e applica più aggregazioni
agg = df.groupby('Team').agg({'Score': ['sum', 'count']})
print("\nSomma e conteggio dei punteggi per team:")
print(agg)
```

- `df.groupby('col')` crea un oggetto GroupBy che divide i dati per i valori unici di 'col'.
- Le funzioni di aggregazione (`mean()`, `sum()`, `count()`, ecc.) calcolano statistiche per ciascun gruppo.
- È possibile raggruppare per più colonne contemporaneamente (`df.groupby(['col1','col2'])`).
- Esito: di solito una Series o DataFrame con indici gerarchici corrispondenti ai gruppi.
- Fondamentale per sintesi dati: ad es. calcolare totali o medie raggruppate.

Pivot Table

Una **pivot table** è una tabella riepilogativa che raggruppa dati in uno schema "matrice" con righe e colonne multiple, aggregando valori. In Pandas si ottiene con `pd.pivot_table(df, index=..., columns=..., values=..., aggfunc=...)`. Questa funzione consente di ricreare tabelle crociate simili a Excel. In pratica si specifica l'indice (righe), le colonne, il campo da aggregare e la funzione di aggregazione (ad es. somma, media). Ad esempio, con `pd.pivot_table(df, values='Value',`

`index='City', columns='Category', aggfunc='sum')` si ottiene la somma dei valori per ogni combinazione di città e categoria. GeeksforGeeks osserva che `pivot_table()` *"allows us to create a pivot table to summarize and aggregate data"* [25]. È utile con dataset grandi: si possono anche gestire i NaN con `fill_value` e aggiungere margini (totali).

```
import pandas as pd

df = pd.DataFrame({
    'City': ['Roma', 'Roma', 'Milano', 'Milano', 'Bologna'],
    'Category': ['A', 'B', 'A', 'B', 'A'],
    'Value': [100, 150, 80, 120, 200]
})

# Crea tabella pivot (somma dei valori per City e Category)
pivot = pd.pivot_table(df, values='Value', index='City', columns='Category',
aggfunc='sum')
print("Tabella pivot (somma):")
print(pivot)

# Con parametro fillna per sostituire NaN
pivot_fill = pd.pivot_table(df, values='Value', index='City',
columns='Category', aggfunc='sum', fill_value=0)
print("\nTabella pivot con fill_value=0:")
print(pivot_fill)
```

- `pd.pivot_table()` crea una tabella stile Excel riassumendo e aggregando i dati [25].
- Parametri chiave: `index` (righe del pivot), `columns` (colonne del pivot), `values` (campo numerico da aggregare), `aggfunc` (media, somma, ecc.).
- Si possono includere più livelli di indice o colonne (liste).
- `fill_value` sostituisce i valori mancanti (NaN) risultanti dal pivot.
- Utile per trasformare dati lunghi in riepiloghi incrociati (sum, average, count, ecc.).

Dati mancanti (NaN)

I valori nulli o **mancanti** in Pandas sono rappresentati come `NaN` (Not a Number, tipo float). Gestire i NaN è fondamentale per analizzare i dati puliti. Pandas offre metodi dedicati: con `df.dropna()` si eliminano righe (o colonne) contenenti NaN, mentre `df.fillna(valore)` riempie i NaN con un valore specifico (come 0 o la media di una colonna). Come evidenziato nella guida, *"i DataFrame forniscono metodi per gestire i valori mancanti o NaN, inclusa la rimozione o il riempimento"* [26]. Spesso si calcola la media di una colonna (ignorando i NaN) e si usa con `fillna()`. Ecco un esempio: `df.dropna()` cancella tutte le righe con almeno un NaN, mentre `df.fillna(df.mean())` sostituisce ogni NaN con la media della colonna corrispondente.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'A': [1, np.nan, 3],
    'B': [4, 5, np.nan]
```

```
})
print("DataFrame originale con NaN:")
print(df)

# Rimuovi righe con NaN
df_drop = df.dropna()
print("\nDopo dropna():")
print(df_drop)

# Sostituisci NaN con 0
df_fill = df.fillna(0)
print("\nDopo fillna(0):")
print(df_fill)

# Sostituisci NaN con media delle colonne
df_mean = df.fillna(df.mean())
print("\nDopo fillna con media delle colonne:")
print(df_mean)
```

- `dropna()`: elimina righe (o colonne, con `axis=1`) che contengono NaN.
- `fillna(valore)`: sostituisce i NaN con il `valore` scelto (es. 0, media, mediana, o metodo di forward/backfill).
- Utilità: evitiamo errori negli algoritmi che non gestiscono bene i NaN e completiamo il dataset prima di calcoli statistici.
- Verifiche: `df.isna()` o `df.notna()` individuano valori NaN nei dati.

Statistiche descrittive

Pandas offre funzioni rapide per ottenere statistiche riassuntive di un DataFrame o di una Series. Ad esempio, `df.mean()`, `df.std()`, `df.min()`, `df.max()` restituiscono rispettivamente media, deviazione standard, minimo e massimo per ciascuna colonna numerica. Il metodo `df.describe()` è particolarmente comodo: genera un sommario completo delle statistiche di *central tendency* e *dispersione* per il DataFrame [27]. Secondo la documentazione, `describe()` *"genera statistiche descrittive che includono tendenza centrale, dispersione e forma della distribuzione"* [27]. Nel caso numerico, fornisce count, mean, std, min, percentili (25%, 50%, 75%) e max. In un tutorial Pandas, vengono citati metodi come `.mean()` e `.std()` proprio per il calcolo rapido delle statistiche fondamentali.

```
import pandas as pd

df = pd.DataFrame({
    'A': [10, 20, 30, 40],
    'B': [5, 5, 5, 5]
})
print("Media colonna A:", df['A'].mean())
print("Deviazione standard colonna A:", df['A'].std())

# Statistiche descrittive riassuntive
```

```
print("\nStatistiche descrittive con describe():")
print(df.describe())
```

- `df['col'].mean()` e `df['col'].std()` calcolano media e deviazione standard di una colonna.
- `df.count()` conta i valori non nulli; `df.min()`, `df.max()`, `df.median()`, ecc. forniscono altre statistiche basilari.
- `df.describe()` riepiloga count, mean, std, min, percentili (25%, 50%, 75%), max per ogni colonna numerica [27].
- Questi metodi ignorano automaticamente i NaN nei calcoli.
- Utile per individuare dati anomali e comprendere la distribuzione delle variabili.

Ordinamento, ridenominazione e trasformazioni

Infine, Pandas consente di **ordinare** facilmente i dati e di trasformare le etichette. Per ordinare un DataFrame in base ai valori di una colonna si usa `df.sort_values(by='col')`; la documentazione spiega infatti che `sort_values()` *"ordina lungo l'asse specificato in base ai valori"* [28]. Si può scegliere l'ordine crescente/decrescente (`ascending=False`) e posizionare i NaN all'inizio o fine (`na_position`). Per ordinare per indice si usa `df.sort_index()`. Per **rinominare** colonne o righe si utilizza `df.rename()`: ad esempio `df.rename(columns={'A':'alpha','B':'beta'})` restituisce un nuovo DataFrame con colonne rinominate [18]. Infine, le **trasformazioni** dei dati possono avvenire con metodi come `apply()` o `applymap()`. Ad esempio, `df['A'].apply(lambda x: x*10)` moltiplica ogni elemento di A per 10. Anche le funzioni per la manipolazione di stringhe o date possono essere applicate alle colonne tramite `.str` o `.dt`.

```
import pandas as pd

df = pd.DataFrame({
    'C': [3, 1, 2],
    'D': ['x', 'y', 'z']
}, index=[2, 0, 1])
print("DataFrame originale:")
print(df)

# Ordinamento per colonna 'C'
df_sorted = df.sort_values(by='C')
print("\nOrdinato per 'C':")
print(df_sorted)

# Rinomina colonne
df_renamed = df.rename(columns={'C': 'ColonnaC', 'D': 'ColonnaD'})
print("\nColonne rinominate:")
print(df_renamed)

# Trasformazione valori di colonna con apply
df['C_times_10'] = df['C'].apply(lambda x: x*10)
```

```
print("\nValori di C moltiplicati per 10 (con apply):")
print(df)
```

- **Ordinamento**: `df.sort_values(by='col')` ordina il DataFrame in base a una (o più) colonne [28]. Con `ascending=False` otteniamo ordine decrescente.
- **Ridenominazione**: `df.rename(columns={vecchio:nuovo})` cambia i nomi delle colonne (o `index={}` per le righe) [18].
- **Trasformazioni**: `df.apply()` applica una funzione per righe o colonne; `df.applymap()` applica elemento-per-elemento.
- Possiamo anche modificare il tipo delle colonne con `df.astype()` o gestire formati di stringhe/datario con `.str` e `.dt`.

---

[1] [2] [3] [4] 12. Ambienti Virtuali e Pacchetti — Documentazione Python 3.13.7
https://docs.python.org/it/3.13/tutorial/venv.html

[5] [6] [7] Che cos'è un notebook Jupyter?
https://www.databricks.com/it/glossary/jupyter-notebook

[8] NumPy - Wikipedia
https://it.wikipedia.org/wiki/NumPy

[9] Tutorial Numpy: Array & Broadcasting - La scienza dei dati
https://lascienzadeidati.altervista.org/tutorial-numpy-array-broadcasting/

[10] [12] [13] Pandas: la libreria Python per l'Analisi dei Dati | Visualitics
https://visualitics.it/pandas-python/

[11] [14] [17] [26] DataFrame vs Series in Pandas - GeeksforGeeks
https://www.geeksforgeeks.org/pandas/dataframe-vs-series-in-pandas/

[15] Pandas Read CSV
https://www.w3schools.com/python/pandas/pandas_csv.asp

[16] pandas.read_excel — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html

[18] pandas.DataFrame.rename — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rename.html

[19] [20] [21] [22] [23] Python basics: unire due DataFrame di Pandas | 1week4
https://www.1week4.com/it/python/pandas-dataframe-merge/

[24] Python Pandas : Aggregazione dati con GroupBy – Appunti di programmazione
https://antoiovi.wordpress.com/python-pandas-aggregazione-dati-con-groupby/

[25] Pandas.pivot_table() - Python - GeeksforGeeks
https://www.geeksforgeeks.org/python/python-pandas-pivot_table/

[27] pandas.DataFrame.describe — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.describe.html

[28] pandas.DataFrame.sort_values — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_values.html