



SVILUPPO APPLICAZIONI E ANALISI DATI CON PYTHON

Pandas Avanzato

Filtrri, Merge e Operazioni Aggiuntive



Indice della Lezione

- Filtri Avanzati: Booleani, `isin()`, `query()`
- Merge e Join: Inner, Outer, Left, Right
- Concatenazione (`concat`) e `append`
- Operazioni su Colonne: `apply()`, `map()`
- Pivot Table e Reshape
- Time Series e Operazioni Temporali
- Caso Studio Completo



Filtri Avanzati: Operatori Booleani

```
# Filtri con AND (&) e OR (|)  
# Attenzione: usare & e |, non 'and' e 'or'  
  
# AND: Entrambe le condizioni vere  
risultato = df[(df['Età'] > 25) & (df['Città'] == 'Roma')]  
  
# OR: Almeno una condizione vera  
risultato = df[(df['Stipendio'] > 3000) | (df['Esperienza'] > 10)]  
  
# NOT (~): Negazione  
risultato = df[~(df['Reparto'] == 'IT')] # Non IT  
  
# Combinazioni complesse  
risultato = df[  
    ((df['Età'] > 30) & (df['Città'] == 'Milano')) |  
    ((df['Stipendio'] > 4000) & (df['Esperienza'] >= 5))  
]
```



Filtri: isin() e between()

```
# isin() - Valori in una lista
regioni = ['Lazio', 'Lombardia', 'Campania']
df_filtrato = df[df['Regione'].isin(regioni)]  
  
# between() - Intervallo di valori (inclusi i limiti)
df_stipendi = df[df['Stipendio'].between(2000, 4000)]  
  
# String methods - Ricerca su stringhe
df_tech = df[df['Ruolo'].str.contains('Developer')]
df_maiuscolo = df[df['Nome'].str.upper() == 'MARIO']  
  
# isna() e notna()
df_completo = df[df['Telefono'].notna()] # Solo chi ha telefono
```



Query Method (Sintassi SQL-like)

```
# .query() permette sintassi simile a SQL
# Più leggibile per filtri complessi

# Senza query (Pandas style)
df1 = df[(df['Età'] > 30) & (df['Città'] == 'Roma')]

# Con query (SQL style)
df2 = df.query('Età > 30 and Città == "Roma"')

# Usando variabili locali con @
limite_age = 25
df3 = df.query('Età > @limite_age')

# Risultato identico, ma più leggibile!
```



Merge: Inner Join

```
import pandas as pd

# Due DataFrame da combinare
studenti = pd.DataFrame({
    'ID': [1, 2, 3],
    'Nome': ['Mario', 'Luigi', 'Peach']
} )

voti = pd.DataFrame({
    'ID': [2, 3, 4],           # ID 4 non è in studenti
    'Matematica': [8, 9, 7]
} )

# INNER JOIN (default): solo ID comuni
risultato = pd.merge(studenti, voti, on='ID')
# Risultato: solo Mario (2) e Peach (3), Luigi (1) e ID 4 scartati
```



Merge: LEFT, RIGHT, OUTER Join

```
# LEFT JOIN: tutti da sinistra + match da destra
left = pd.merge(studenti, voti, on='ID', how='left')
# Risultato: Mario(1, NaN), Luigi(2, 8), Peach(3, 9)

# RIGHT JOIN: tutti da destra + match da sinistra
right = pd.merge(studenti, voti, on='ID', how='right')
# Risultato: Luigi(2, 8), Peach(3, 9), ID4(4, 7)

# OUTER JOIN: Tutti da entrambi
outer = pd.merge(studenti, voti, on='ID', how='outer')
# Risultato: Mario(1, NaN), Luigi(2, 8), Peach(3, 9), ID4(NaN, 7)

# Merge su colonne con nomi diversi
risultato = pd.merge(df1, df2, left_on='ID_Stu', right_on='ID_Voti')
```



Concat: Concatenare DataFrame

```
# concat() - Accatastare DataFrame

df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

# Concatenare verticalmente (impilare)
df_stacked = pd.concat([df1, df2], ignore_index=True)

# Concatenare orizzontalmente (lato a lato)
df_side = pd.concat([df1, df2], axis=1)

# Differenza tra concat e merge:
# concat() = impila senza cercare corrispondenze
# merge() = combina cercando chiavi comuni
```



Apply: Applicare funzioni a colonne

```
# apply() - Applica funzione a ogni elemento/riga

# Funzione personalizzata
def categorizza_voto(voto):
    if voto >= 8:
        return 'Eccellente'
    elif voto >= 6:
        return 'Sufficiente'
    else:
        return 'Insufficiente'

# Applicare a una colonna
df['Categoria'] = df['Voto'].apply(categorizza_voto)

# Lambda per operazioni semplici
df['Doppio'] = df['Numero'].apply(lambda x: x * 2)

# apply() su intere righe
df['Somma'] = df.apply(lambda riga: riga['A'] + riga['B'], axis=1)

# map() - Solo per Series, più veloce per mappature semplici
mappa = {1: 'uno', 2: 'due', 3: 'tre'}
df['Testo'] = df['Numero'].map(mappa)
```



Pivot Table: Rimodellare dati

```
# pivot_table() - Come le tavole pivot di Excel

df = pd.DataFrame({
    'Mese': ['Gen', 'Gen', 'Feb', 'Feb'],
    'Città': ['Roma', 'Milano', 'Roma', 'Milano'],
    'Vendite': [100, 200, 150, 250]
})

# Tabella pivot
pivot = df.pivot_table(
    values='Vendite',           # Cosa sommare/aggredire
    index='Mese',               # Righe
    columns='Città',             # Colonne
    aggfunc='sum'                # Funzione aggregazione
)

# Risultato:
# Città      Milano  Roma
# Mese
# Feb        250     150
# Gen        200     100
```



Melt: Trasformare Pivot in Long Format

```
# melt() - Opposto di pivot_table()
# Trasforma da wide a long format

# Dati in wide format
df_wide = pd.DataFrame({
    'Nome': ['Mario', 'Luigi'],
    'Matematica': [8, 7],
    'Italiano': [6, 8]
})

# Trasformare in long format
df_long = df_wide.melt(
    id_vars=['Nome'],           # Colonne da mantenere
    value_vars=['Matematica', 'Italiano'], # Colonne da "fondere"
    var_name='Materia',        # Nome per le colonne originali
    value_name='Voto'          # Nome per i valori
)
```



Gestione Indici: set_index / reset_index

```
# set_index() - Rendere una colonna come indice
df_indexed = df.set_index('ID')

# reset_index() - Convertire indice in colonna
df_reset = df_indexed.reset_index()

# drop=True per scartare il vecchio indice
df_reset = df_indexed.reset_index(drop=True)

# MultiIndex - Indici multipli
df_multi = df.set_index(['Città', 'Reparto'])

# Accesso con tupl per MultiIndex
valore = df_multi.loc[('Roma', 'IT'), 'Stipendio']

# Operazioni su indice
df.index.is_unique      # Controlla unicità
df.index.duplicated()   # Mostra duplicati
```



Time Series: Operazioni temporali

```
# Convertire colonna in datetime
df['Data'] = pd.to_datetime(df['Data'])

# Estrarre componenti temporali
df['Anno'] = df['Data'].dt.year
df['Mese'] = df['Data'].dt.month
df['Giorno_Settimana'] = df['Data'].dt.day_name()

# Ordinare per data
df_sorted = df.sort_values('Data')

# Filtrare per intervallo temporale
inizio = pd.to_datetime('2025-01-01')
fine = pd.to_datetime('2025-06-30')
df_trimestre = df[(df['Data'] >= inizio) & (df['Data'] <= fine)]

# set_index con data per operazioni temporali veloci
df_ts = df.set_index('Data')
# Ora puoi usare df_ts.loc['2025-01':'2025-03']
```



Rolling Window: Media Mobile

```
# rolling() - Calcola metriche su finestre mobili

# Media mobile su 7 giorni
df['Media_7gg'] = df['Vendite'].rolling(window=7).mean()

# Somma su 30 giorni
df['Somma_30gg'] = df['Vendite'].rolling(window=30).sum()

# Deviazione standard mobile
df['Std_5gg'] = df['Vendite'].rolling(window=5).std()

# Utile per visualizzare tendenze e eliminare rumore
# Notare che i primi (window-1) valori saranno NaN
```



Nan: Gestione avanzata

```
# Forward fill - Copia valore precedente
df['Colonna'] = df['Colonna'].fillna(method='ffill')

# Backward fill - Copia valore successivo
df['Colonna'] = df['Colonna'].fillna(method='bfill')

# Interpolazione - Calcola valori intermedi
df['Colonna'] = df['Colonna'].interpolate()

# Riempire con media per gruppo
df['Colonna'] = df.groupby('Gruppo')['Colonna'].transform(
    lambda x: x.fillna(x.mean()))
)

# Contare Nan
count_nan = df.isna().sum()
```



Duplicati: Identificazione e Rimozione

```
# Identificare duplicati
duplicati = df.duplicated()    # Booleano per ogni riga
duplicati_su_id = df.duplicated(subset=['ID'])

# Rimuovere duplicati
df_clean = df.drop_duplicates()

# Mantenere primo/ultimo occorrenza
df_first = df.drop_duplicates(subset=['ID'], keep='first')
df_last = df.drop_duplicates(subset=['ID'], keep='last')

# Numero di duplicati per colonna
for col in df.columns:
    print(f'{col}: {df[col].duplicated().sum()} duplicati')
```



Operazioni su Stringhe (.str)

```
# .str accessor per operazioni su stringhe

# Trasformazione caso
df['Nome_Upper'] = df['Nome'].str.upper()
df['Nome_Lower'] = df['Nome'].str.lower()

# Ricerca e sostituzione
df['Email_Anonima'] = df['Email'].str.replace(r'@.*', '@***', regex=True)

# Split
df[['Nome', 'Cognome']] = df['NomeCompleto'].str.split(' ', expand=True)

# Contains (case-insensitive)
developer = df[df['Ruolo'].str.contains('dev', case=False)]

# Estrazione con regex
df['Anno'] = df['Data_Testo'].str.extract(r'(\d{4})')

# Lunghezza stringa
df['Lunghezza_Nome'] = df['Nome'].str.len()
```



Rename Colonne e Reindex

```
# Rinominare colonne (metodo 1: dizionario)
df = df.rename(columns={
    'ID_Vecchio': 'ID',
    'Nome_Prod': 'Prodotto'
} )

# Rinominare colonne (metodo 2: funzione)
df = df.rename(columns=str.upper) # Tutte maiuscole

# Rinominare indice
df = df.rename(index={0: 'riga_0', 1: 'riga_1'})

# Reindex - Riordina o aggiunge righe
nuovo_indice = [0, 2, 4, 6, 100]
df_reindexed = df.reindex(nuovo_indice)
# 100 avrà NaN in tutte le colonne
```



Sort Avanzato: Ordinamento Multi-livello

```
# sort_values() - Ordinamento su una colonna
df_sorted = df.sort_values('Stipendio', ascending=False)

# Multi-colonna: ordinare per Città, poi Stipendio
df_multi = df.sort_values(
    by=['Città', 'Stipendio'],
    ascending=[True, False] # Città ASC, Stipendio DESC
)

# sort_index() - Ordinare per indice
df_index_sorted = df.sort_index()

# Ordine personalizzato (categoria)
ordine = ['Basso', 'Medio', 'Alto']
df['Categoria'] = pd.Categorical(
    df['Categoria'],
    categories=ordine,
    ordered=True
)
df_cat = df.sort_values('Categoria')
```



Caso Studio: E-Commerce avanzato (Part 1)

```
import pandas as pd

# Caricamento dati
clienti = pd.read_csv('clienti.csv')
ordini = pd.read_csv('ordini.csv')
prodotti = pd.read_csv('prodotti.csv')

# Merge: ordini + clienti (LEFT: tutti gli ordini)
vendite = pd.merge(ordini, clienti, on='ClienteID', how='left')

# Merge: vendite + prodotti
vendite = pd.merge(vendite, prodotti, on='ProdottoID')

# Filtri avanzati: ordini recenti e di valore > 100€
vendite['Data'] = pd.to_datetime(vendite['Data'])
recenti = vendite[
    (vendite['Data'] >= '2025-01-01') &
    (vendite['Importo'] > 100)
]
```



Caso Studio: E-Commerce avanzato (Part 2)

```
# Pivot Table: Vendite per Città e Categoria Prodotto
vendite_pivot = pd.pivot_table(
    recenti,
    values='Importo',
    index='Città',
    columns='Categoria',
    aggfunc='sum',
    fill_value=0
)

# Aggiungere colonna commissione
recenti['Commissione'] = recenti['Importo'] * 0.15

# Raggruppare per cliente: totale speso
per_cliente = recenti.groupby('NomeCliente').agg({
    'Importo': 'sum',
    'ClienteID': 'count' # Numero ordini
}).rename(columns={'ClienteID': 'NumOrdini'})

# Esportare risultati
per_cliente.to_csv('report_clienti.csv')
```



Best Practices e Performance

- Usa `pd.merge()` con indici come chiavi per velocità
- Preferisci `query()` per filtri leggibili complessi
- Usa `apply()` con cautela (lento): preferisci vettorizzazione
- Chunked reading per file giganti: `pd.read_csv(..., chunksize=10000)`
- Usa tipi di dato efficienti (category, int32 vs int64)
- Valida sempre i merge con controlli post-operazione



Strumenti Consigliati

Jupyter Notebook

Esplorazione interattiva, ideale per analisi dati

→ Data Science

PyCharm Prof

Progetti grandi, debugging avanzato, integrazioni

→ Production



Riepilogo Lezione

- ✓ Filtri avanzati: &, |, ~, isin(), query()
- ✓ Merge/Join: inner, left, right, outer
- ✓ Apply, map, pivot_table, melt
- ✓ Time series, rolling windows, string operations



Domande?

Approfondimenti:

- NumPy: Array multidimensionali e operazioni numeriche
- Matplotlib e Seaborn: Visualizzazioni avanzate
- Machine Learning: Scikit-learn basics