



Python

GIORNO 4 - Pandas -Leggere i dati da file, filtrare i dati, funzioni base

Agenda:

- Pandas - Funzioni per leggere dati da files
- Pandas - Attributi e metodi base
- Pandas - Filtrare i dati



Perché?

- I dati tabulari possono essere archiviati in **diversi formati** ed è importante sapere come leggerli in Pandas al di là di queste differenze
- Una prima **analisi esplorativa** può essere affrontata grazie a delle funzioni base già incluse in Pandas
- Saper **filtrare** i dati in base a condizioni complesse e dinamiche ci permette poi di effettuare analisi approfondite con una maggiore facilità e precisione.

Alla fine di questa lezione saprete:

- Caricare qualsiasi formato di dati tabulari in Pandas**
- Valutare caratteristiche generali dei vostri dataframe grazie a funzioni base di Pandas**
- Filtrare in base a index e colonne usando `.loc[]` e `.iloc[]`**
- Costruire dei filtri**
- Effettuare query sui DataFrame**

Leggere i dati da file

I formati per archiviare dati tabulari su un computer sono strutture standardizzate che organizzano le informazioni, consentendo una facile manipolazione e interpretazione dei dati da parte delle applicazioni software

Nella maggior parte dei casi la struttura di organizzazione è quella suddivisa in righe e colonne, ma esistono anche altri casi particolari.

Leggere i dati da file

Alcuni tra i principali sistemi di organizzazione dei dati sono:

- CSV** (Comma-Separated Values): È un formato di file testuale che utilizza virgole (principalmente) per separare i valori delle colonne
- TSV** (Tab-Separated Values): Simile al CSV, ma utilizza il carattere di tabulazione (un numero di whitespace predefiniti) per separare i valori delle colonne
- XLS/XLSX** (Excel Spreadsheet): Sono formati di file proprietari di Microsoft Excel per fogli di calcolo
- JSON** (JavaScript Object Notation): Non è esclusivamente tabulare, ma può essere utilizzato per rappresentare dati tabulari. È un formato di file che utilizza una sintassi basata su coppie chiave-valore e array.

Dato che ognuno di questi formati organizza i dati strutturandoli in modo diverso rispetto agli altri, Pandas ci fornisce delle funzioni specifiche per gestirli. Andiamo a vedere le principali

`pd.read_csv()`

Per leggere un file in formato CSV utilizziamo la funzione `read_csv()`. Questa funzione accetta svariati parametri, tra cui:

- filepath**: il path del file da leggere, espresso come stringa o come URL
- sepstr**: il carattere che delimita i dati, settato di default come `" , "` può essere cambiato.
Per leggere un file `.tsv`, ad esempio, utilizzeremo `\t`
- header**: il numero di riga che contiene le etichette delle colonne e che definisce l'inizio dei dati
- names**: una lista di nomi per le colonne
- index_col**: colonna da usare come indice della tabella
- usecols**: subset di colonne da leggere
- dtype**: datatypes da applicare a tutto il `DataFrame` o alle singole colonne

L'unico parametro obbligatorio è `filepath=`, gli altri sono tutti opzionali e ce ne sono molti di più: ci riferiamo alla documentazione per completezza. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

pd.read_csv()

```
import pandas as pd

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
column_names = ["sepal_length", "sepal_width", "petal_length", "petal_width", "species"]
iris = pd.read_csv(url, names=column_names)

print(iris)
```

| | sepal_length | sepal_width | petal_length | petal_width | species |
|-----|--------------|-------------|--------------|-------------|-------------|
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |
| ... | ... | ... | ... | ... | ... |

`pd.read_excel()`

Pandas può leggere file di Excel gestendo i fogli multipli, le formule e le varie particolarità di questi file grazie alla funzione `read_excel()`. Ecco alcuni parametri:

- io**: il path del file da leggere, espresso come stringa o come URL.
- sheet_name**: il nome, o gli indici, dei fogli da caricare. Carica il primo di default, ma possiamo passargli anche una lista o `None` per caricarli tutti
- header**: il numero di riga che contiene le etichette delle colonne e che definisce l'inizio dei dati.
- names**: una lista di nomi per le colonne.
- index_col**: colonna da usare come indice della tabella.
- usecols**: subset di colonne da leggere.
- dtype**: datatypes da applicare a tutto il `DataFrame` o alle singole colonne.

Documentazione:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

```
pd.read_excel()
```

```
import pandas as pd

url = "https://go.microsoft.com/fwlink/?LinkID=521962"
manufacturing = pd.read_excel(url)

print(manufacturing)
```

`pd.read_json()`

Per caricare dati in formato JSON in un DataFrame usiamo la funzione `read_json()`. Ecco alcuni parametri:

- path_or_buf**: li path del file da leggere, espresso come stringa o come URL
- orient**: formattazione del JSON
- typ**: il tipo di oggetto da recuperare (**o** DataFrame **o** Series)
- dtype**: datatypes da applicare a tutto il DataFrame **o alle singole colonne**
- convert_axes**: effettuare la conversione automatica degli assi a dei `dtype`

Documentazione:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html

Encoding

Tutti i metodi che abbiamo analizzato prima accettano un parametro: il parametro `encoding=`.

L'encoding è il processo di conversione dei dati da una forma a un'altra per la memorizzazione o la trasmissione. In informatica, si riferisce principalmente alla conversione dei caratteri in byte per la memorizzazione in memoria o nei file

Per tutti i metodi visti è settato di default come "`utf-8`", che rappresenta l'encoding standard per la maggior parte dei caratteri. A seconda dei dati che stiamo leggendo può essere necessario modificare questo parametro: troviamo un elenco esaustivo degli encoding presenti in Python in questa documentazione

<https://docs.python.org/3/library/codecs.html#standard-encodings>

Attributi base

Una volta che abbiamo caricato i nostri dati in un `DataFrame`, avremo un oggetto che ha degli **attributi** (cioè delle caratteristiche) e dei **metodi** (cioè delle funzioni) proprie. Cominciamo col vedere alcuni attributi base, che ci permettono di scoprire alcune caratteristiche dei nostri dati.

- `.shape`: Restituisce una tupla con due elementi che rappresentano le dimensioni del `DataFrame` (**numero di righe, numero di colonne**)

- `.columns`: Restituisce un elenco delle etichette delle colonne nel `DataFrame`

- `.index`: Restituisce un oggetto che rappresenta l'indice (etichette delle righe) del `DataFrame`

- `.dtypes`: Restituisce un oggetto che contiene i tipi di dati di ciascuna colonna del `DataFrame`

Attributi base

Possiamo pensare agli attributi come a delle variabili incapsulate all'interno del nostro DataFrame, i cui valori quindi cambiano a seconda del DataFrame stesso. Per esempio, indaghiamo la tabella *factresellersales* del nostro DB AdventureWorksDW:

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df.shape  
✓ 3.2s  
(57851, 14)
```

Tramite l'attributo `.shape` abbiamo accesso ad una tuple che contiene due valori interi: il primo è il numero delle righe del nostro DataFrame, il secondo invece è il numero delle colonne. La tabella *factresellersales* ha dunque 14 colonne e 57851 righe.

Attributi base

Attraverso l'attributo `.columns` invece accediamo ai nomi delle colonne:

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.columns
✓ 2.8s
Index(['SalesOrderNumber', 'SalesOrderLineNumber', 'OrderDate', 'DueDate',
       'ShipDate', 'ProductKey', 'ResellerKey', 'PromotionKey', 'EmployeeKey',
       'SalesTerritoryKey', 'OrderQuantity', 'UnitPrice', 'TotalProductCost',
       'SalesAmount'],
      dtype='object')
```

ATTENZIONE: `columns` non contiene una lista, ma un oggetto del tipo

`pandas.core.indexes.base.Index` che può non avere gli stessi metodi e attributi di una lista.

Possiamo sempre trasformarlo in una lista con il type casting `list(df.columns)` o utilizzando il metodo `.tolist()`

```
1 print(type(df.columns), type(list(df.columns)), type(df.columns.tolist()))
✓ 0.0s
<class 'pandas.core.indexes.base.Index'> <class 'list'> <class 'list'>
```

Attributi base

L'attributo `.index`, invece, ci dà accesso agli indici del DataFrame.

Anche qui, come per le colonne, facciamo attenzione al tipo di dato contenuto nell'attributo, che in questo caso è un `RangeIndex`, ma potrebbe essere diverso a seconda dei dati contenuti nel DataFrame.

Come sempre, per vedere tutte le varie possibilità facciamo riferimento alla documentazione:
<https://pandas.pydata.org/docs/reference/api/pandas.Index.html>

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.index
✓ 2.4s
RangeIndex(start=0, stop=57851, step=1)
```

Attributi base

Nell'attributo `.dtypes` è contenuta una `Series` i cui indici sono i nomi delle colonne e i valori sono il datatype che contengono.

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df.dtypes  
✓ 2.5s  
  
SalesOrderNumber      object  
SalesOrderLineNumber   int64  
OrderDate             object  
DueDate               object  
ShipDate              object  
ProductKey            int64  
ResellerKey           int64  
PromotionKey          int64  
EmployeeKey           int64  
SalesTerritoryKey     int64  
OrderQuantity         int64  
UnitPrice              float64  
TotalProductCost      float64  
SalesAmount            float64  
dtype: object
```

Ricordiamoci che Pandas è basato su NumPy: questo comporta che, al suo interno, rappresenta i dati tramite le tipologie implementate da NumPy e non quelle native di Python.

Metodi base

Oltre agli attributi che ci permettono un'overview dei dati, abbiamo anche dei **metodi**. Se gli attributi sono delle caratteristiche degli oggetti, i metodi sono delle funzioni che applicano delle operazioni sull'oggetto stesso. Vediamo alcuni dei metodi base:

- `.info()`: Stampa una panoramica concisa del DataFrame
- `.describe()`: Restituisce statistiche descrittive per ogni colonna nel DataFrame
- `.head(n)`: Restituisce le prime *n* righe del DataFrame
- `.tail(n)`: Restituisce le ultime *n* righe del DataFrame
- `.sample(n)`: Restituisce *n* righe casuali del DataFrame
- `.nunique()`: Restituisce il numero di valori unici per ciascuna colonna nel DataFrame

Metodi base

`pd.DataFrame.info()` -> non restituisce alcun valore ma stampa le informazioni

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.info()
✓ 2.3s

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 57851 entries, 0 to 57850
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   SalesOrderNumber    57851 non-null   object 
 1   SalesOrderLineNumber 57851 non-null   int64  
 2   OrderDate          57851 non-null   object 
 3   DueDate            57851 non-null   object 
 4   ShipDate           57165 non-null   object 
 5   ProductKey         57851 non-null   int64  
 6   ResellerKey        57851 non-null   int64  
 7   PromotionKey       57851 non-null   int64  
 8   EmployeeKey        57851 non-null   int64  
 9   SalesTerritoryKey  57851 non-null   int64  
 10  OrderQuantity      57851 non-null   int64  
 11  UnitPrice          57851 non-null   float64 
 12  TotalProductCost   57775 non-null   float64 
 13  SalesAmount         57851 non-null   float64 
dtypes: float64(3), int64(7), object(4)
memory usage: 6.2+ MB
```

Metodi base

`pd.DataFrame.describe()` -> restituisce un DataFrame

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.describe()
✓ 2.0s
```

| | SalesOrderLineNumber | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount |
|-------|----------------------|--------------|--------------|--------------|--------------|-------------------|---------------|--------------|------------------|--------------|
| count | 57851.000000 | 57851.000000 | 57851.000000 | 57851.000000 | 57851.000000 | 57851.000000 | 57851.000000 | 57851.000000 | 57775.000000 | 57851.000000 |
| mean | 16.397469 | 408.636705 | 340.742978 | 1.274239 | 286.131735 | 4.554424 | 3.528271 | 446.388513 | 1322.850666 | 1340.487981 |
| std | 13.101347 | 113.665645 | 205.493906 | 1.630692 | 4.521832 | 2.412247 | 3.035766 | 521.880164 | 2170.219318 | 2151.753062 |
| min | 1.000000 | 212.000000 | 1.000000 | 1.000000 | 272.000000 | 1.000000 | 1.000000 | 1.330000 | 0.860000 | 1.370000 |
| 25% | 6.000000 | 326.000000 | 166.000000 | 1.000000 | 283.000000 | 3.000000 | 2.000000 | 34.930000 | 97.150000 | 129.560000 |
| 50% | 13.000000 | 401.000000 | 327.000000 | 1.000000 | 285.000000 | 4.000000 | 3.000000 | 214.240000 | 461.440000 | 469.790000 |
| 75% | 24.000000 | 491.000000 | 514.000000 | 1.000000 | 290.000000 | 6.000000 | 4.000000 | 672.290000 | 1510.300000 | 1516.160000 |
| max | 72.000000 | 606.000000 | 701.000000 | 16.000000 | 296.000000 | 10.000000 | 44.000000 | 2146.960000 | 38530.390000 | 30993.040000 |

Metodi base

`pd.DataFrame.head(n)` -> restituisce un DataFrame

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.head(5)
✓ 2.0s
```

| | SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount |
|---|------------------|----------------------|------------|------------|------------|------------|-------------|--------------|-------------|-------------------|---------------|-----------|------------------|-------------|
| 0 | SO43659 | 1 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 349 | 676 | 1 | 285 | 5 | 1 | 2024.99 | NaN | 2024.99 |
| 1 | SO43659 | 2 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 350 | 676 | 1 | 285 | 5 | 3 | 2024.99 | NaN | 6074.97 |
| 2 | SO43659 | 3 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 351 | 676 | 1 | 285 | 5 | 1 | 2024.99 | NaN | 2024.99 |
| 3 | SO43659 | 4 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 344 | 676 | 1 | 285 | 5 | 1 | 2039.99 | NaN | 2039.99 |
| 4 | SO43659 | 5 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 345 | 676 | 1 | 285 | 5 | 1 | 2039.99 | NaN | 2039.99 |

`pd.DataFrame.tail(n)` -> restituisce un DataFrame

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.tail(5)
✓ 2.5s
```

| | SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount |
|-------|------------------|----------------------|------------|------------|----------|------------|-------------|--------------|-------------|-------------------|---------------|-----------|------------------|-------------|
| 57846 | SO69565 | 3 | 2020-05-31 | 2020-06-10 | None | 565 | 8 | 1 | 285 | 5 | 2 | 445.41 | 922.89 | 890.82 |
| 57847 | SO69565 | 4 | 2020-05-31 | 2020-06-10 | None | 572 | 8 | 1 | 285 | 5 | 1 | 445.41 | 461.44 | 445.41 |
| 57848 | SO69565 | 5 | 2020-05-31 | 2020-06-10 | None | 560 | 8 | 1 | 285 | 5 | 1 | 728.91 | 755.15 | 728.91 |
| 57849 | SO69565 | 6 | 2020-05-31 | 2020-06-10 | None | 555 | 8 | 1 | 285 | 5 | 1 | 63.90 | 47.29 | 63.90 |
| 57850 | SO69565 | 7 | 2020-05-31 | 2020-06-10 | None | 484 | 8 | 1 | 285 | 5 | 1 | 4.77 | 2.97 | 4.77 |

Metodi base

`pd.DataFrame.sample(n)` -> restituisce un DataFrame

```

1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.sample(10)
✓ 2.3s

```

| | SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount |
|-------|------------------|----------------------|------------|------------|------------|------------|-------------|--------------|-------------|-------------------|---------------|-----------|------------------|-------------|
| 6719 | SO46042 | 5 | 2018-05-09 | 2018-05-19 | 2018-05-16 | 310 | 1 | 1 | 286 | 1 | 1 | 2146.96 | 2171.29 | 2146.96 |
| 20664 | SO48392 | 35 | 2018-12-29 | 2019-01-08 | 2019-01-05 | 263 | 54 | 1 | 281 | 2 | 2 | 202.33 | 374.31 | 404.66 |
| 27868 | SO50294 | 12 | 2019-05-25 | 2019-06-04 | 2019-06-01 | 213 | 312 | 1 | 282 | 4 | 2 | 20.19 | 27.76 | 40.38 |
| 23883 | SO49485 | 26 | 2019-03-12 | 2019-03-22 | 2019-03-19 | 458 | 155 | 1 | 291 | 6 | 4 | 44.99 | 123.73 | 179.96 |
| 40662 | SO55269 | 12 | 2019-10-12 | 2019-10-22 | 2019-10-19 | 545 | 290 | 1 | 289 | 1 | 5 | 24.29 | 89.89 | 121.45 |
| 24837 | SO49827 | 18 | 2019-04-03 | 2019-04-13 | 2019-04-10 | 263 | 299 | 1 | 291 | 6 | 3 | 202.33 | 561.47 | 606.99 |
| 495 | SO43861 | 8 | 2017-08-10 | 2017-08-20 | 2017-08-17 | 314 | 584 | 1 | 285 | 5 | 1 | 2146.96 | 2171.29 | 2146.96 |
| 48741 | SO61244 | 26 | 2020-01-24 | 2020-02-03 | 2020-01-31 | 490 | 110 | 1 | 286 | 1 | 4 | 32.39 | 166.29 | 129.56 |
| 7106 | SO46065 | 2 | 2018-05-18 | 2018-05-28 | 2018-05-25 | 350 | 335 | 1 | 288 | 6 | 1 | 2024.99 | 1898.09 | 2024.99 |
| 11284 | SO46982 | 1 | 2018-08-13 | 2018-08-23 | 2018-08-20 | 401 | 157 | 1 | 290 | 7 | 1 | 65.60 | 48.55 | 65.60 |

Metodi base

pd.DataFrame.nunique() -> restituisce una Series

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.nunique()
✓ 1.9s
SalesOrderNumber      3616
SalesOrderLineNumber    72
OrderDate              990
DueDate                990
ShipDate                983
ProductKey              334
ResellerKey              632
PromotionKey              12
EmployeeKey              17
SalesTerritoryKey          10
OrderQuantity              41
UnitPrice                230
TotalProductCost          1410
SalesAmount                1411
dtype: int64
```

Metodi base

Esistono anche **metodi che si applicano alle singole colonne** e non all'intero DataFrame.

I DataFrame sono strutture bidimensionali che hanno quindi due indicizzazioni: una "verticale", cioè l'**indice**, che identifica ogni riga, e una "orizzontale", cioè le **colonne**.

Abbiamo visto che, con le Series, possiamo accedere ad un indice specifico utilizzando le parentesi quadre. I DataFrame, invece, utilizzano lo stesso comportamento per accedere ad una colonna specifica.

Per esempio:

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df["OrderDate"]
✓ 1.9s
0    2017-07-01
1    2017-07-01
2    2017-07-01
3    2017-07-01
4    2017-07-01
...
57846   2020-05-31
57847   2020-05-31
57848   2020-05-31
57849   2020-05-31
57850   2020-05-31
Name: OrderDate, Length: 57851, dtype: object
```

Metodi base

Altri due metodi molto utili che si applicano alle singole colonne sono `.unique()` e `.value_counts()`.

- `pd.DataFrame[column].unique()` -> restituisce un ndarray contenente i valori unici presenti nella colonna

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df["SalesTerritoryKey"].unique()  
✓ 1.7s  
array([ 5,  6,  4,  1,  3,  2, 10,  7,  9,  8])
```

Metodi base

- ❑ `pd.DataFrame[column].value_counts()` -> restituisce una Series i cui indici sono i valori unici della colonna in relazione al conteggio del numero di occorrenze di ogni valore

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df["SalesTerritoryKey"].value_counts()  
✓ 2.1s  
  
SalesTerritoryKey  
4    12826  
6    10934  
1     7446  
5     5742  
2     5610  
3     5527  
7     3422  
10    3242  
8     1679  
9     1423  
Name: count, dtype: int64
```

Metodi base

- `pd.DataFrame.set_index(column)` -> Trasforma una colonna del nostro DataFrame in indice.

```

1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.set_index("OrderDate")

```

✓ 3.9s

| | SalesOrderNumber | SalesOrderLineNumber | DueDate | ShipDate | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount |
|------------|------------------|----------------------|------------|------------|------------|-------------|--------------|-------------|-------------------|---------------|-----------|------------------|-------------|
| OrderDate | | | | | | | | | | | | | |
| 2017-07-01 | SO43659 | 1 | 2017-07-11 | 2017-07-08 | 349 | 676 | 1 | 285 | 5 | 1 | 2024.99 | NaN | 2024.99 |
| 2017-07-01 | SO43659 | 2 | 2017-07-11 | 2017-07-08 | 350 | 676 | 1 | 285 | 5 | 3 | 2024.99 | NaN | 6074.97 |
| 2017-07-01 | SO43659 | 3 | 2017-07-11 | 2017-07-08 | 351 | 676 | 1 | 285 | 5 | 1 | 2024.99 | NaN | 2024.99 |
| 2017-07-01 | SO43659 | 4 | 2017-07-11 | 2017-07-08 | 344 | 676 | 1 | 285 | 5 | 1 | 2039.99 | NaN | 2039.99 |
| 2017-07-01 | SO43659 | 5 | 2017-07-11 | 2017-07-08 | 345 | 676 | 1 | 285 | 5 | 1 | 2039.99 | NaN | 2039.99 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 2020-05-31 | SO69565 | 3 | 2020-06-10 | None | 565 | 8 | 1 | 285 | 5 | 2 | 445.41 | 922.89 | 890.82 |
| 2020-05-31 | SO69565 | 4 | 2020-06-10 | None | 572 | 8 | 1 | 285 | 5 | 1 | 445.41 | 461.44 | 445.41 |
| 2020-05-31 | SO69565 | 5 | 2020-06-10 | None | 560 | 8 | 1 | 285 | 5 | 1 | 728.91 | 755.15 | 728.91 |
| 2020-05-31 | SO69565 | 6 | 2020-06-10 | None | 555 | 8 | 1 | 285 | 5 | 1 | 63.90 | 47.29 | 63.90 |
| 2020-05-31 | SO69565 | 7 | 2020-06-10 | None | 484 | 8 | 1 | 285 | 5 | 1 | 4.77 | 2.97 | 4.77 |

Metodi base

- Se andiamo a stampare il DataFrame, ci accorgeremo che non è stato modificato

- In generale, i metodi che trasformano i DataFrame restituiscono un nuovo DataFrame, che possiamo stampare a video o memorizzare in una variabile

- Se vogliamo invece rendere permanenti le modifiche sul DataFrame originario, dobbiamo assegnare il risultato ad una nuova variabile, oppure alla stessa (**sovrascrittura**)

Filtrare i dati

Come abbiamo visto, possiamo accedere ad una colonna specifica di un DataFrame tramite le parentesi quadre e il nome della colonna.

Se invece vogliamo accedere a delle righe specifiche, dobbiamo specificare la posizione numerica d'inizio e di fine tramite la procedura dello slicing, ad esempio:

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df[0:4]
```

✓ 1.8s

| | SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount |
|---|------------------|----------------------|------------|------------|------------|------------|-------------|--------------|-------------|-------------------|---------------|-----------|------------------|-------------|
| 0 | SO43659 | 1 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 349 | 676 | 1 | 285 | 5 | 1 | 2024.99 | NaN | 2024.99 |
| 1 | SO43659 | 2 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 350 | 676 | 1 | 285 | 5 | 3 | 2024.99 | NaN | 6074.97 |
| 2 | SO43659 | 3 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 351 | 676 | 1 | 285 | 5 | 1 | 2024.99 | NaN | 2024.99 |
| 3 | SO43659 | 4 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 344 | 676 | 1 | 285 | 5 | 1 | 2039.99 | NaN | 2039.99 |

Questa notazione però può essere ambigua e portare ad errori. Per accedere a dei subset specifici di dati con più sicurezza possiamo usare .loc[] e .iloc[]

Filtrare i dati

- `.loc[]` è un attributo utilizzato per selezionare righe e colonne in base a etichette (nomi di riga o colonna)
- `.iloc[]` è utilizzato per selezionare righe e colonne in base agli indici (indexing o slicing)

ATTENZIONE: è facile confondersi dato che spesso i nomi delle righe (gli indici) sono appunto dei numeri interi, e quindi potremmo avere l'impressione che `.loc[]` lavori sugli indici in maniera posizionale ma non è così. Ricordatevi l'esempio degli indici numerici disordinati delle `Series`.

`.loc[]` e `.iloc[]` **restituiscono sempre il tipo di dato più semplice:** se è un dato monodimensionale (una singola colonna o una singola riga) restituiscono una `Series`; se è bidimensionale restituiscono un `DataFrame`

Filtrare i dati

Per esempio: se voglio selezionare le righe il cui indice è il numero intero 5 userò

```
pd.DataFrame.loc[5]
```

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.loc[5]
✓ 3.7s
```

| SalesOrderNumber | SO43659 |
|----------------------|------------------|
| SalesOrderLineNumber | 6 |
| OrderDate | 2017-07-01 |
| DueDate | 2017-07-11 |
| ShipDate | 2017-07-08 |
| ProductKey | 346 |
| ResellerKey | 676 |
| PromotionKey | 1 |
| EmployeeKey | 285 |
| SalesTerritoryKey | 5 |
| OrderQuantity | 2 |
| UnitPrice | 2039.99 |
| TotalProductCost | NaN |
| SalesAmount | 4079.98 |
| Name: | 5, dtype: object |

Filtrare i dati

Per esempio: se voglio selezionare le righe il cui indice è compreso tra 5 e 10, userò

```
pd.DataFrame.loc[5:10]
```

```
1 query = """SELECT *
2 FROM factresellersales"""
3 df = pd.read_sql(query, db_engine)
4 df.loc[5:10]
✓ 1.8s
```

| | SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount |
|----|------------------|----------------------|------------|------------|------------|------------|-------------|--------------|-------------|-------------------|---------------|-----------|------------------|-------------|
| 5 | SO43659 | 6 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 346 | 676 | 1 | 285 | 5 | 2 | 2039.99 | NaN | 4079.98 |
| 6 | SO43659 | 7 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 347 | 676 | 1 | 285 | 5 | 1 | 2039.99 | NaN | 2039.99 |
| 7 | SO43659 | 8 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 229 | 676 | 1 | 285 | 5 | 3 | 28.84 | NaN | 86.52 |
| 8 | SO43659 | 9 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 235 | 676 | 1 | 285 | 5 | 1 | 28.84 | NaN | 28.84 |
| 9 | SO43659 | 10 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 218 | 676 | 1 | 285 | 5 | 6 | 5.70 | NaN | 34.20 |
| 10 | SO43659 | 11 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 223 | 676 | 1 | 285 | 5 | 2 | 5.19 | NaN | 10.38 |

Filtrare i dati

Dato che i DataFrame sono strutture dati bidimensionali, possiamo usare `.loc[]` anche per accedere alle colonne, specificando il loro nome in seconda posizione, dopo gli indici, all'interno delle parentesi quadre. Per esempio:

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df.loc[5, "SalesOrderNumber"]  
✓ 2.6s  
'S043659'
```

In questo modo stiamo accedendo al dato specifico che corrisponde all'intersezione della riga 5 con la colonna “*SalesOrderNumber*”.

Filtrare i dati

Possiamo anche accedere ad un **subset di colonne**, inserendo invece dell'etichetta (il nome della colonna) singola una lista di etichette:

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df.loc[5, ["SalesOrderNumber", "OrderDate"]]  
✓ 2.7s  
  
SalesOrderNumber      SO43659  
OrderDate            2017-07-01  
Name: 5, dtype: object
```

E possiamo combinare un subset di righe e un subset di colonne:

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df.loc[5:10, ["SalesOrderNumber", "OrderDate"]]  
✓ 2.3s  
  
SalesOrderNumber  OrderDate  
5           SO43659  2017-07-01  
6           SO43659  2017-07-01  
7           SO43659  2017-07-01  
8           SO43659  2017-07-01  
9           SO43659  2017-07-01  
10          SO43659  2017-07-01
```

Filtrare i dati

QUIZZZONE



Gli indici del nostro DataFrame sono cambiati: ora sono delle combinazioni random di numeri e lettere:

| SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate | ProductKey | ResellerKey |
|------------------|----------------------|-----------|------------|------------|------------|-------------|
| mblatGhElz | SO43659 | 1 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 349 |
| 8cA0DKiX74 | SO43659 | 2 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 350 |
| Eeeh1O9g1u | SO43659 | 3 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 351 |
| Yx19yFTM6H | SO43659 | 4 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 344 |
| PM4BdhKBau | SO43659 | 5 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 345 |
| ... | ... | ... | ... | ... | ... | ... |
| xkBXdFlnIL | SO69565 | 3 | 2020-05-31 | 2020-06-10 | None | 565 |
| Y9oQ6PDAfy | SO69565 | 4 | 2020-05-31 | 2020-06-10 | None | 572 |
| AsSg6G7ZtH | SO69565 | 5 | 2020-05-31 | 2020-06-10 | None | 560 |
| LWFMCyBRvW | SO69565 | 6 | 2020-05-31 | 2020-06-10 | None | 555 |
| dMftx8zqQX | SO69565 | 7 | 2020-05-31 | 2020-06-10 | None | 484 |

Possiamo comunque accedere alle righe utilizzando
`df.loc[0:5]`?

Filtrare i dati

QUIZZZONE



E se estraessimo dieci righe a caso con il metodo
.sample(), assegnando il risultato ad una nuova
variabile?

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df_sample = df.sample(10)  
5 df_sample
```

✓ 2.8s

| SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount | |
|------------------|----------------------|-----------|------------|------------|------------|-------------|--------------|-------------|-------------------|---------------|-----------|------------------|-------------|---------|
| 45043 | SO58921 | 2 | 2019-12-03 | 2019-12-13 | 2019-12-10 | 481 | 306 | 1 | 283 | 3 | 6 | 5.39 | 20.17 | 32.34 |
| 35854 | SO51857 | 15 | 2019-08-29 | 2019-09-08 | 2019-09-05 | 503 | 85 | 1 | 292 | 7 | 2 | 200.05 | 399.70 | 400.10 |
| 38512 | SO53554 | 16 | 2019-09-19 | 2019-09-29 | 2019-09-26 | 472 | 195 | 1 | 294 | 9 | 4 | 38.10 | 95.00 | 152.40 |
| 55858 | SO69442 | 11 | 2020-05-09 | 2020-05-19 | 2020-05-16 | 355 | 233 | 1 | 283 | 2 | 2 | 1391.99 | 2531.24 | 2783.98 |
| 9119 | SO46631 | 6 | 2018-07-13 | 2018-07-23 | 2018-07-20 | 456 | 17 | 1 | 281 | 5 | 4 | 44.99 | 123.73 | 179.96 |
| 7499 | SO46099 | 4 | 2018-05-29 | 2018-06-08 | 2018-06-05 | 342 | 45 | 1 | 285 | 5 | 5 | 419.46 | 2065.73 | 2097.30 |
| 31041 | SO51120 | 28 | 2019-07-15 | 2019-07-25 | 2019-07-22 | 550 | 196 | 1 | 288 | 10 | 1 | 149.87 | 136.79 | 149.87 |
| 18493 | SO48057 | 4 | 2018-11-20 | 2018-11-30 | 2018-11-27 | 379 | 343 | 1 | 293 | 1 | 3 | 1308.94 | 3962.05 | 3926.82 |
| 49528 | SO63171 | 24 | 2020-02-10 | 2020-02-20 | 2020-02-17 | 506 | 230 | 1 | 295 | 8 | 2 | 200.05 | 399.70 | 400.10 |
| 11298 | SO46984 | 5 | 2018-08-14 | 2018-08-24 | 2018-08-21 | 224 | 677 | 1 | 291 | 6 | 1 | 5.19 | 5.23 | 5.19 |

Filtrare i dati

Per selezionare i dati in base alla posizione delle righe e delle colonne e non al loro “nome” o “etichetta” usiamo `.iloc[]`.

Otterrò quindi le prime cinque righe del mio `DataFrame` in questo modo:

```
df.iloc[0:5]
```

✓ 0.0s

| | SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate | ProductKey | ResellerKey | PromotionKey | EmployeeKey | SalesTerritoryKey | OrderQuantity | UnitPrice | TotalProductCost | SalesAmount |
|------------|------------------|----------------------|------------|------------|------------|------------|-------------|--------------|-------------|-------------------|---------------|-----------|------------------|-------------|
| mlatGhElz | SO43659 | 1 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 349 | 676 | 1 | 285 | 5 | 1 | 2024.99 | NaN | 2024.99 |
| 8cA0DKiX74 | SO43659 | 2 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 350 | 676 | 1 | 285 | 5 | 3 | 2024.99 | NaN | 6074.97 |
| Eehh1O9g1u | SO43659 | 3 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 351 | 676 | 1 | 285 | 5 | 1 | 2024.99 | NaN | 2024.99 |
| Yx19yFTM6H | SO43659 | 4 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 344 | 676 | 1 | 285 | 5 | 1 | 2039.99 | NaN | 2039.99 |
| PM4BdhKBau | SO43659 | 5 | 2017-07-01 | 2017-07-11 | 2017-07-08 | 345 | 676 | 1 | 285 | 5 | 1 | 2039.99 | NaN | 2039.99 |

Filtrare i dati

`.iloc[]` funziona seguendo la stessa logica di `.loc[]`, ma utilizza degli indici posizionali 0-based. Possiamo dunque accedere anche alle colonne, ma invece che specificare le loro etichette, o i loro nomi, dovrò specificare la loro **posizione**, sempre secondo la logica dell'indexing o dello slicing.

Ad esempio, per accedere alle prime cinque righe e le prime cinque colonne potrò usare:

```
df.iloc[0:5, 0:5]
```

✓ 0.0s

| | SalesOrderNumber | SalesOrderLineNumber | OrderDate | DueDate | ShipDate |
|------------|------------------|----------------------|------------|------------|------------|
| mblatGhElz | SO43659 | 1 | 2017-07-01 | 2017-07-11 | 2017-07-08 |
| 8cA0DKiX74 | SO43659 | 2 | 2017-07-01 | 2017-07-11 | 2017-07-08 |
| Eeeh1O9g1u | SO43659 | 3 | 2017-07-01 | 2017-07-11 | 2017-07-08 |
| Yx19yFTM6H | SO43659 | 4 | 2017-07-01 | 2017-07-11 | 2017-07-08 |
| PM4BdhKBau | SO43659 | 5 | 2017-07-01 | 2017-07-11 | 2017-07-08 |

Filtrare i dati

Per selezionare tutte le righe, o tutte le colonne, usiamo il carattere “ `:` ”, da solo (che, come visto nella lezione sulle liste, usato così significa dall'inizio alla fine), dove solitamente scriverebbero l'indice numerico, nel caso di `.iloc[]`, oppure l'etichetta, nel caso di `.loc[]`

Per esempio:

```
1 query = """SELECT *  
2 FROM factresellersales"""  
3 df = pd.read_sql(query, db_engine)  
4 df.loc[:, ["SalesOrderNumber", "OrderDate"]]  
✓ 2.2s
```

| | SalesOrderNumber | OrderDate |
|-------|------------------|------------|
| 0 | SO43659 | 2017-07-01 |
| 1 | SO43659 | 2017-07-01 |
| 2 | SO43659 | 2017-07-01 |
| 3 | SO43659 | 2017-07-01 |
| 4 | SO43659 | 2017-07-01 |
| ... | ... | ... |
| 57846 | SO69565 | 2020-05-31 |
| 57847 | SO69565 | 2020-05-31 |
| 57848 | SO69565 | 2020-05-31 |
| 57849 | SO69565 | 2020-05-31 |
| 57850 | SO69565 | 2020-05-31 |

57851 rows × 2 columns

Filtrare i dati

Ricapitolando, `.loc[]` si può usare, per esempio:

```
df.loc[ etichetta ]    # seleziona una riga  
df.loc[ etichetta1:etichetta2 ]  # seleziona delle righe consecutive (range)  
df.loc[ [etichetta1, etichetta2, etichetta3] ]  # seleziona delle righe specifiche
```

```
df.loc[ :, colonna ]    # seleziona una colonna  
df.loc[ :, colonna1:colonna2 ]  # seleziona delle colonne consecutive (range)  
df.loc[ :, [colonna1, colonna2, colonna3] ]  # seleziona delle colonne specifiche
```

Per avere un codice più chiaro si possono usare delle variabili intermedie:

```
righe_selezionate = [etichetta1, etichetta2, etichetta3, etichetta4]  
colonne_selezionate = [colonna1, colonna2]  
df.loc[ righe_selezionate, colonne_selezionate ]  # selez. 4 righe e 2 colonne specifiche
```

Filtrare i dati

Ricapitolando, `.iloc[]` si può usare, per esempio:

```
df.iloc[ indice_riga ]    # seleziona una riga (INDEXING)
df.iloc[ indice_riga1:indice_riga2 ]  # seleziona delle righe consecutive (SLICING)
df.iloc[ [indice_riga1, indice_riga2, indice_riga3] ]  # seleziona delle righe specifiche

df.iloc[ :, indice_colonna ]  # seleziona una colonna (INDEXING)
df.iloc[ :, indice_colonna1:indice_colonna2 ]  # seleziona colonne consecutive (SLICING)
df.iloc[ :, [indice_colonna1, indice_colonna2] ]  # seleziona delle colonne specifiche
```

Per avere un codice più chiaro si possono usare delle variabili intermedie:

```
righe_selezionate = [indice_riga1, indice_riga2, indice_riga3, indice_riga4]
colonne_selezionate = [indice_colonna1, indice_colonna2]
df.iloc[ righe_selezionate, colonne_selezionate ]  # selez. 4 righe e 2 colonne specifiche
```

Filtrare i dati

.loc[] e .iloc[] permettono non solo di selezionare dei subset di dati, ma di applicare dei **filtri specifici**:

Pandas si basa su NumPy e, grazie a questa derivazione, permette di applicare la procedura del **masking**.

Per esempio, andiamo a vedere la tabella *dimcustomer*.

```
query = "SELECT * FROM dimcustomer"

df = pd.read_sql(query, db_engine)

print(df)
```

Filtrare i dati

Filtrare i dati

Possiamo creare un filtro sulla colonna `CommuteDistance` tramite un operatore di confronto, in modo da scoprire quali clienti devono spostarsi di non più di un miglio, selezionando il valore `0-1 Miles`:

La nostra variabile `filtro` conterrà una `Series` i cui indici sono gli stessi indici del `DataFrame` e i cui valori sono il risultato del confronto che abbiamo espresso.

Andando ad applicare questa maschera/filtro tramite `.loc[]`, otterremo dal `DataFrame` soltanto le righe che rispettano la condizione espressa, cioè soltanto quelle righe che nella `Series` contenuta nella variabile `filtro` corrispondono a `True`

```
filtro = df.loc[:, "CommuteDistance"] == "0-1 Miles"  
  
df.loc[filtro]
```

Filtrare i dati

Filtrare i dati

Possiamo anche **filtrare usando più parametri di selezione, utilizzando la logica booleana mediante `and`, `or` e `not`.** La sintassi di Pandas vuole però che vengano espressi tramite questi simboli:

- **&**: `and`
- **|**: `or`
- **~**: `not`

Filtrare i dati

Per esempio, prendiamo un subset di colonne da *dimcustomer* e filtriemo i dati per ottenere le donne che lavorano nel management, cioè estraiamo soltanto quelle righe per cui Gender è uguale a "F" e EnglishOccupation è uguale a "Management" :

```
filtro_gender = df.loc[:, "Gender"] == "F"

filtro_occupation = df.loc[:, "EnglishOccupation"] == "Management"

filtro = filtro_gender & filtro_occupation

df.loc[filtro]
```

Filtrare i dati

Filtrare i dati

Se invece voglio negare la condizione che ho espresso in una delle maschere, ad esempio per estrarre soltanto gli uomini in ruolo di management, posso farlo senza modificare la maschera stessa e utilizzando l'operatore `~` all'interno di `.loc[]`

```
filtro_gender = df.loc[:, "Gender"] == "F"

filtro_occupation = df.loc[:, "EnglishOccupation"] == "Management"

filtro = ~filtro_gender & filtro_occupation

df.loc[filtro]
```

Filtrare i dati

Filtrare i dati

L'operatore **| (or)** serve invece a selezionare le righe per cui almeno una delle condizioni che abbiamo espresso sia vera, per esempio:

```
filtro_skilled = df.loc[:, "EnglishOccupation"] == "Skilled Manual"

filtro_manual = df.loc[:, "EnglishOccupation"] == "Manual"

filtro = filtro_skilled | filtro_manual

df.loc[filtro]
```

Filtrare i dati

| | CustomerKey | Title | FirstName | MiddleName | LastName | NameStyle | BirthDate | Suffix | Gender | EmailAddress | ... | EnglishOccupation |
|-------|-------------|-------|-----------|------------|----------|-----------|------------|--------|--------|---------------------------------|-----|-------------------|
| 15 | 11015 | None | Chloe | None | Young | 0 | 1984-08-26 | None | F | chloe23@adventure-works.com | ... | Skilled Manual |
| 16 | 11016 | None | Wyatt | L | Hill | 0 | 1984-10-25 | None | M | wyatt32@adventure-works.com | ... | Skilled Manual |
| 17 | 11017 | None | Shannon | None | Wang | 0 | 1949-12-24 | None | F | shannon1@adventure-works.com | ... | Skilled Manual |
| 19 | 11019 | None | Luke | L | Lal | 0 | 1983-09-04 | None | M | luke18@adventure-works.com | ... | Skilled Manual |
| 20 | 11020 | None | Jordan | C | King | 0 | 1984-03-19 | None | M | jordan73@adventure-works.com | ... | Skilled Manual |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 18473 | 29473 | None | Carmen | J | Subram | 0 | 1965-11-19 | None | F | carmen16@adventure-works.com | ... | Manual |
| 18474 | 29474 | None | Jaime | B | Raje | 0 | 1965-04-02 | None | M | jaime36@adventure-works.com | ... | Manual |
| 18475 | 29475 | None | Jared | A | Ward | 0 | 1976-03-18 | None | M | jared6@adventure-works.com | ... | Manual |
| 18476 | 29476 | None | Elizabeth | None | Bradley | 0 | 1964-12-30 | None | F | elizabeth30@adventure-works.com | ... | Manual |

Filtrare i dati

Un metodo per filtrare selezionando solo alcuni dati di interesse è il metodo `.isin()`, equivalente del comando `in`
Permette di valutare se i dati di una colonna si trovano tra quelli all'interno di una lista

```
cognomi_di_interesse = ["Yang", "Torres", "Lu"]
filtro_cognomi = df.loc[:, "LastName"].isin(cognomi_di_interesse)
filtro_colonne = ["CustomerKey", "FirstName", "LastName"]
df.loc[filtro_cognomi, filtro_colonne]
```

| | CustomerKey | FirstName | LastName |
|-------|-------------|-----------|----------|
| 0 | 11000 | Jon | Yang |
| 2 | 11002 | Ruben | Torres |
| 11 | 11011 | Curtis | Lu |
| 47 | 11047 | Jaclyn | Lu |
| 110 | 11110 | Curtis | Yang |
| ... | ... | ... | ... |
| 17723 | 28723 | Ryan | Yang |
| 17732 | 28732 | Bianca | Yang |
| 17925 | 28925 | Warren | Lu |
| 18243 | 29243 | Jared | Torres |
| 18394 | 29394 | Raquel | Torres |

Filtrare i dati

Esistono molti altri modi per filtrare i dati di un DataFrame: sostanzialmente **ogni confronto che restituisca un booleano può essere utilizzato come filtro.**

Per questo motivo non è possibile farne un elenco completo, ma riferitevi sempre alle documentazioni ufficiali e, ragionando sulla tipologia di dato su cui state lavorando, troverete tutte le risposte alle domande che potrebbero sorgervi.

Oppure chiedete ai vostri Teacher o TA :)

Ordinamento

Per ordinare un DataFrame si usa il metodo `.sort_values()`.

Leggiamo un dataset di modelli di smartphone; per semplicità selezioniamo solo le prime righe con l'attributo `nrows=` e alcune colonne con l'attributo `usecols=`:

```
import pandas as pd

url = "https://raw.githubusercontent.com/YBI-Foundation/Dataset/refs/heads/main/MobilePriceRange.csv"

selected_cols = ["DualSim", "MobileWeight", "ScreenHeight", "ScreenWidth", "PriceRange"]

df = pd.read_csv(url, nrows=10, usecols=selected_cols)

print(df)
```

Ordinamento

| | DualSim | MobileWeight | ScreenHeight | ScreenWidth | PriceRange |
|---|---------|--------------|--------------|-------------|------------|
| 0 | 0 | 188 | 9 | 7 | Medium |
| 1 | 1 | 136 | 17 | 3 | High |
| 2 | 1 | 145 | 11 | 2 | High |
| 3 | 0 | 131 | 16 | 8 | High |
| 4 | 0 | 141 | 8 | 2 | Medium |
| 5 | 1 | 164 | 17 | 1 | Medium |
| 6 | 0 | 139 | 13 | 8 | VeryHigh |
| 7 | 1 | 187 | 16 | 3 | Low |
| 8 | 0 | 174 | 17 | 1 | Low |
| 9 | 1 | 93 | 19 | 10 | Low |

Ordinamento

Ordiniamo il DataFrame mediante la colonna con il peso, in ordine decrescente tramite il parametro ascending= (che di default è True):

```
df = df.sort_values("MobileWeight", ascending=False)  
print(df)
```

| | DualSim | MobileWeight | ScreenHeight | ScreenWidth | PriceRange |
|---|---------|--------------|--------------|-------------|------------|
| 0 | 0 | 188 | 9 | 7 | Medium |
| 7 | 1 | 187 | 16 | 3 | Low |
| 8 | 0 | 174 | 17 | 1 | Low |
| 5 | 1 | 164 | 17 | 1 | Medium |
| 2 | 1 | 145 | 11 | 2 | High |
| 4 | 0 | 141 | 8 | 2 | Medium |
| 6 | 0 | 139 | 13 | 8 | VeryHigh |
| 1 | 1 | 136 | 17 | 3 | High |
| 3 | 0 | 131 | 16 | 8 | High |
| 9 | 1 | 93 | 19 | 10 | Low |

Ordinamento

È possibile anche effettuare un ordinamento nidificato, dando in input una lista con le colonne di interesse, ad esempio ordiniamo prima per i modelli con dual SIM, poi per peso:

```
df = df.sort_values(["DualSim", "MobileWeight"], ascending=False)  
print(df)
```

| | DualSim | MobileWeight | ScreenHeight | ScreenWidth | PriceRange |
|---|---------|--------------|--------------|-------------|------------|
| 7 | 1 | 187 | 16 | 3 | Low |
| 5 | 1 | 164 | 17 | 1 | Medium |
| 2 | 1 | 145 | 11 | 2 | High |
| 1 | 1 | 136 | 17 | 3 | High |
| 9 | 1 | 93 | 19 | 10 | Low |
| 0 | 0 | 188 | 9 | 7 | Medium |
| 8 | 0 | 174 | 17 | 1 | Low |
| 4 | 0 | 141 | 8 | 2 | Medium |
| 6 | 0 | 139 | 13 | 8 | VeryHigh |
| 3 | 0 | 131 | 16 | 8 | High |



GRAZIE
EPCODE