



Python **GIORNO 7 - Pandas - EDA e visualizzazioni**

Agenda:

- Pandas - Exploratory Data Analysis

- Seaborn - Visualizzazioni



Perché?

- Uno dei primi step per portare avanti l'analisi di un dataset è effettuare un'esplorazione di base per sapere con che cosa si ha a che fare

- Poter creare delle visualizzazioni è utile in ogni step dell'analisi: dal momento esplorativo alla presentazione degli insight agli stakeholder.

Alla fine di questa lezione saprete:

- Cos'è un EDA e come effettuarla in maniera efficace
- Utilizzare la libreria Seaborn per avere un controllo maggiore sulle vostre visualizzazioni

Exploratory Data Analysis

Il successo di qualsiasi analisi dati inizia con una solida Exploratory Data Analysis (EDA). Questa fase è fondamentale per ottenere insight significativi e preparare il terreno per analisi più avanzate.



Exploratory Data Analysis

Alcune delle funzioni dell'EDA sono:

- **Identificare Pattern:** L'EDA consente di individuare pattern e tendenze nei dati, fornendo una comprensione intuitiva dei comportamenti chiave.
- **Individuare Outlier:** Attraverso grafici e statistiche descrittive, l'EDA aiuta a individuare dati anomali o outlier che potrebbero influenzare negativamente le analisi successive.
- **Comprendere la Distribuzione dei Dati:** L'EDA fornisce strumenti per visualizzare la distribuzione dei dati in modo chiaro e accessibile.
- **Preparare i Dati:** L'EDA aiuta a identificare e affrontare sfide come i dati mancanti, garantendo che l'analisi successiva sia basata su dati di alta qualità.

Exploratory Data Analysis

Una serie generale di step da intraprendere durante l'EDA, usando Pandas, può essere:

- Prima overview dei dati
- Visualizzazione delle informazioni generali sul dataset
- Analisi delle colonne numeriche
- Analisi delle colonne categoriche
- Individuazione dei dati nulli

Effettuiamo ognuno di questi step e prima di passare al successivo chiediamoci sempre:

Che informazioni generali sto acquisendo?

EDA - prima overview dei dati

Sul nostro database AdventureWorksDW c'è una vista chiamata *"vw_factresellersales_denorm"*, composta da unioni tra varie tabelle al fine di denormalizzare i dati presenti in *"factresellersales"*.

La prima cosa che facciamo, dopo aver caricato i dati, è stampare le prime righe del DataFrame.

```
query = """SELECT *
FROM vw_factresellersales_denorm"""
df = pd.read_sql(query, db_engine)
```

✓ 4.9s

```
df.head()
```

✓ 0.0s

	SalesOrderNumber	SalesOrderLineNumber	OrderDate	EnglishProductName	EnglishProductSubcategoryName	EnglishProductCategoryName	OrderQuantity	UnitPrice	TotalProductCost	SalesAmount
0	SO43659	1	2017-07-01	Mountain-100 Black, 42	Mountain Bikes	Bikes	1	2024.99	NaN	2024.99
1	SO43659	2	2017-07-01	Mountain-100 Black, 44	Mountain Bikes	Bikes	3	2024.99	NaN	6074.97
2	SO43659	3	2017-07-01	Mountain-100 Black, 48	Mountain Bikes	Bikes	1	2024.99	NaN	2024.99
3	SO43659	4	2017-07-01	Mountain-100 Silver, 38	Mountain Bikes	Bikes	1	2039.99	NaN	2039.99
4	SO43659	5	2017-07-01	Mountain-100 Silver, 42	Mountain Bikes	Bikes	1	2039.99	NaN	2039.99

Possiamo già ottenere delle informazioni generali da queste poche righe?

EDA - visualizzazione delle informazioni generali

Il secondo step consiste nell'aumentare la granularità dell'overview cercando **informazioni generali per ogni singola colonna**. Utilizziamo il metodo `.info()` di Pandas:

```
df.info()
✓ 0.0s

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 57851 entries, 0 to 57850
Data columns (total 10 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   EnglishProductName   57851 non-null   object 
 4   EnglishProductSubcategoryName 57851 non-null   object 
 5   EnglishProductCategoryName 57851 non-null   object 
 6   OrderQuantity       57851 non-null   int64  
 7   UnitPrice          57851 non-null   float64 
 8   TotalProductCost    57775 non-null   float64 
 9   SalesAmount         57851 non-null   float64 
dtypes: float64(3), int64(2), object(5)
memory usage: 4.4+ MB
```

EDA - analisi delle colonne numeriche

Il nostro DataFrame contiene sia colonne numeriche che categoriche. Effettuiamo una prima indagine di quelle numeriche tramite il metodo `.describe()`, che restituisce un DataFrame contenente alcuni descrittori statistici di base escludendo automaticamente le colonne categoriche.

```
df.describe()
✓ 0.05
```

	SalesOrderLineNumber	OrderQuantity	UnitPrice	TotalProductCost	SalesAmount
count	57851.000000	57851.000000	57851.000000	57775.000000	57851.000000
mean	16.397469	3.528271	446.388513	1322.850666	1340.487981
std	13.101347	3.035766	521.880164	2170.219318	2151.753062
min	1.000000	1.000000	1.330000	0.860000	1.370000
25%	6.000000	2.000000	34.930000	97.150000	129.560000
50%	13.000000	3.000000	214.240000	461.440000	469.790000
75%	24.000000	4.000000	672.290000	1510.300000	1516.160000
max	72.000000	44.000000	2146.960000	38530.390000	30993.040000

EDA - analisi delle colonne categoriche

Una prima analisi delle colonne categoriche può essere fatta indagando sulla quantità di dati unici contenuti e sulla loro frequenza. Pandas non ci offre un metodo pre-confezionato, ma possiamo ottenere il risultato grazie ad una combinazione di strumenti che già conosciamo:

- Otteniamo l'elenco delle colonne del DataFrame accedendo all'attributo `.columns`
- Iteriamo con un `for loop` sulle colonne
- Per ogni colonna:
 - Verifichiamo la tipologia dei dati accedendo all'attributo `.dtype` e utilizzando l'istruzione condizionale `if`. Se la colonna contiene dati di tipo “`object`”:
 - Contiamo quanti valori unici contiene con il metodo `.nunique()`
 - Verifichiamo la frequenza di questi valori con il metodo `.value_counts()`

EDA - analisi delle colonne categoriche

Traduciamo in Python, aggiungendo dei print() per stampare i dati a schermo:

```
# Iteriamo ogni colonna del DataFrame
for column in df.columns:
    # Verifichiamo se è di tipo categorico
    if df[column].dtype=="object":
        print(f"{column} column contains categorical data.")
        # Conteggio dei valori unici presenti nella colonna
        unique_count = df[column].nunique()
        print(f"{column} contains {unique_count} unique values")
        # Frequenza di ogni valore unico
        all_count = df[column].value_counts()
        print(f"Distribution of every unique value is as such:")
        print(all_count)
        print()
```

EDA - analisi delle colonne categoriche

L'output che otteniamo è molto lungo, perché alcune colonne contengono un numero elevato di valori univoci che generano una Series composta da tanti elementi quando chiamiamo il metodo `.value_counts()`

```
SalesOrderNumber column contains categorical data.  
SalesOrderNumber contains 3616 unique values  
Distribution of every unique value is as such:  
SalesOrderNumber  
S051721    72  
S051739    72  
S051160    71  
S053465    71  
S047355    68  
...  
S046632    1  
S061259    1  
S061260    1  
S061261    1  
S061268    1  
Name: count, Length: 3616, dtype: int64
```

```
OrderDate column contains categorical data.  
OrderDate contains 990 unique values  
Distribution of every unique value is as such:  
OrderDate  
2019-09-02    301  
2019-12-03    290  
2019-07-15    285  
2019-12-01    262  
...  
Clothing      11745  
Accessories   4826  
Name: count, dtype: int64
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...

EDA - analisi delle colonne categoriche

QUIZZONE



Possiamo modificare il codice che abbiamo usato per ottenere un output più leggibile?

HINT: Potremmo innestare un'altra istruzione condizionale.

EDA - individuazione dei dati nulli

Ormai conosciamo la problematica rappresentata dai dati nulli e, prima ci interfacciamo ad essi, più la nostra analisi potrà essere accurata. Il metodo `.info()` restituisce un conteggio dei valori non nulli per ogni colonna, ma utilizzando un loop ed il metodo `.isna()` possiamo ottenere un risultato più immediato e leggibile.

```
# Iteriamo ogni colonna del DataFrame
for column in df.columns:
    # Sommiamo i valori nulli (i dati booleani vengono interpretati come 1 e 0)
    nan_count = df[column].isna().sum()
    # Calcoliamo la percentuale
    nan_percentage = round((nan_count/df.shape[0])*100, 2)
    print(f"{column} contains {nan_count} NaN values, {nan_percentage}% of all rows.")
✓ 0.0s

SalesOrderNumber contains 0 NaN values, 0.0% of all rows.
SalesOrderLineNumber contains 0 NaN values, 0.0% of all rows.
OrderDate contains 0 NaN values, 0.0% of all rows.
EnglishProductName contains 0 NaN values, 0.0% of all rows.
EnglishProductSubcategoryName contains 0 NaN values, 0.0% of all rows.
EnglishProductCategoryName contains 0 NaN values, 0.0% of all rows.
OrderQuantity contains 0 NaN values, 0.0% of all rows.
UnitPrice contains 0 NaN values, 0.0% of all rows.
TotalProductCost contains 76 NaN values, 0.13% of all rows.
SalesAmount contains 0 NaN values, 0.0% of all rows.
```

EDA - individuazione dei dati nulli

Un altro metodo per investigare i valori nulli, e allo stesso tempo i valori esistenti all'interno di una singola colonna — specialmente per quanto riguarda le colonne categoriche — è di utilizzare il parametro `dropna=` (settandolo a `False`) del metodo `.value_counts()` che conta anche i valori nulli, e che di default è settato a `True`

```
df.loc[:, "colonna"].value_counts(dropna=False)
```

Un metodo veloce di calcolare tutti i valori nulli espliciti di ogni colonna è di usare `.isna()` assieme a `.sum()`; questo si può fare perché i valori `True` equivalgono al valore `1`, mentre i `False` al valore `0`, quindi il metodo `.sum()` ci dice in effetti quanti valori `True` sono presenti

Per cui, per calcolare velocemente tutti i valori nulli espliciti di un `DataFrame` possiamo scrivere:

```
df.isna().sum()
```

Visualizzazioni

La **visualizzazione dei dati** è una parte fondamentale della data analysis, perché trasforma dati complessi e talvolta astratti in informazioni visivamente comprensibili e significative, sfruttando anche il fatto che gli esseri umani sono molto più bravi nella comprensione delle informazioni visive rispetto a enormi tabelle di dati numerici e/o testuali.

Attraverso grafici, diagrammi e altre rappresentazioni visive, è possibile identificare pattern, tendenze e relazioni che altrimenti potrebbero facilmente sfuggire, e aiuta a individuare anomalie, outlier o errori nei dati, fornendo un feedback immediato sull'integrità e sulla qualità dei dati stessi, contribuendo così a guidare processi decisionali più intelligenti e informate.

Per questi motivi la visualizzazione dei dati consente di comunicare in modo efficace i risultati dell'analisi, facilitando la comprensione e la presa di decisioni informate, anche e soprattutto verso interlocutori che non sono esperti di data analysis; pensate per esempio a come le infografiche siano un modo molto diffuso e apprezzato di divulgare informazioni e dati.

Visualizzazioni

Esempio: lo sfondo "Bliss" di Windows XP

Al centro dell'immagine si apre una vasta collina verde, morbida e lussureggiante, che si estende fino all'orizzonte. La superficie della collina è erbosa, il cui verde vibrante crea una netta linea di divisione con il cielo azzurro sovrastante.

Le linee ondulate della collina conferiscono un senso di movimento e dinamicità alla scena.

Nel cielo azzurro che sovrasta la scena appaiono nuvole bianche, attraverso le quali filtra la luce del sole, creando giochi di luce e ombra sul suolo.



Quale delle due è di più veloce comprensione? In altre parole, quale delle due "arriva" prima?

Visualizzazioni con Pandas

Per visualizzare i dati possiamo utilizzare alcuni metodi dei `DataFrame` che creano automaticamente dei grafici e sono naturalmente ottimizzati per i dati in formato tabellare e con metadati.

Ad esempio, dai `beginner_datasets` possiamo leggere il dataset `population.csv` che contiene l'andamento della popolazione (come percentuale di variazione, e non numero assoluto) di numerosi paesi e aree geografiche dal 1961 al 2015, da cui poi selezioniamo alcuni paesi.

Per avere gli anni sulle righe e i paesi sulle colonne, utilizziamo il metodo `.transpose()` che scambia righe con colonne.

Visualizzazioni con Pandas

```
pop = pd.read_csv("<PATH>/beginner_datasets/population.csv")
pop = pop.set_index("Country Name")
pop = pop.transpose()
pop = pop.loc[:, ["Italy", "Spain", "Portugal",
                  "Germany", "France", "Switzerland"]]
pop.head()
```

Country Name	Italy	Spain	Portugal	Germany	France	Switzerland
1961	0.668383	0.929016	0.805085	0.769854	1.337853	1.978615
1962	0.676623	0.920032	0.721622	0.879432	1.412470	2.535013
1963	0.729553	0.877043	0.403566	0.925875	1.411512	2.137663
1964	0.822624	0.993696	0.055464	0.805141	1.314427	1.654258
1965	0.842109	1.085845	-0.407787	0.853190	1.154839	1.154843

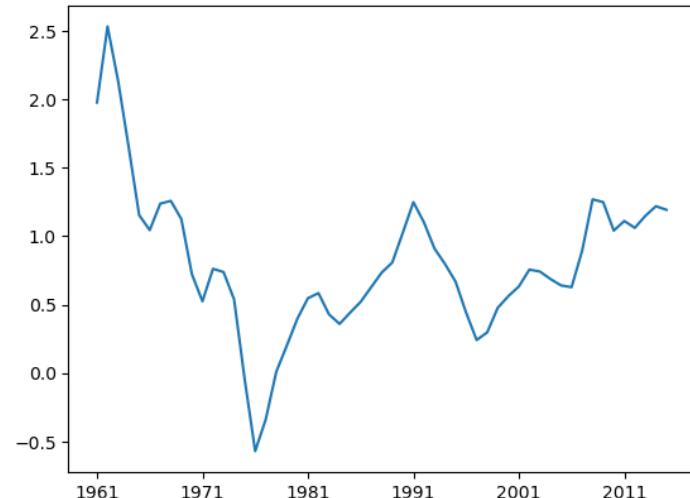
Visualizzazioni con Pandas

Per analizzare l'andamento della popolazione svizzera possiamo visualizzare la colonna relativa:

```
print( pop.Switzerland )
```

Tramite il metodo `.plot()` possiamo però visualizzare immediatamente l'andamento:

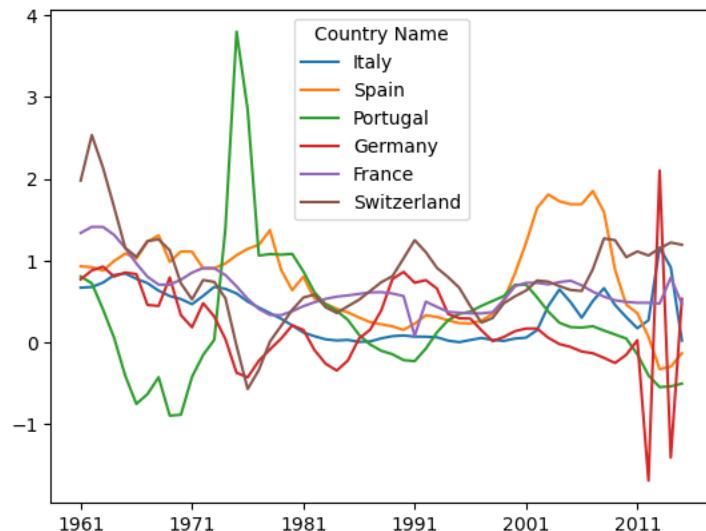
```
pop.Switzerland.plot()
```



Visualizzazioni con Pandas

Se invece utilizziamo il metodo `.plot()` su tutto il DataFrame, automaticamente otterremo l'andamento dei dati di tutte le colonne:

```
pop.plot()
```



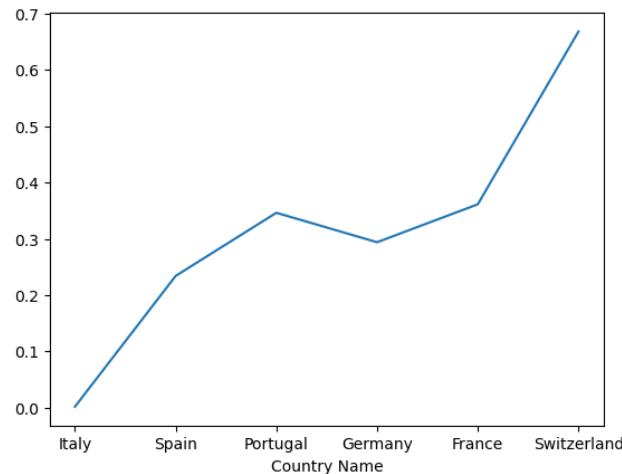
Questo tipo di grafico è definito `lineplot`

Visualizzazioni con Pandas

E se volessimo valutare la differenza di popolazione tra i paesi selezionati per un anno specifico?

Ad esempio, selezioniamo l'anno 1995 e vediamo il grafico relativo:

```
pop.loc["1995"].plot()
```

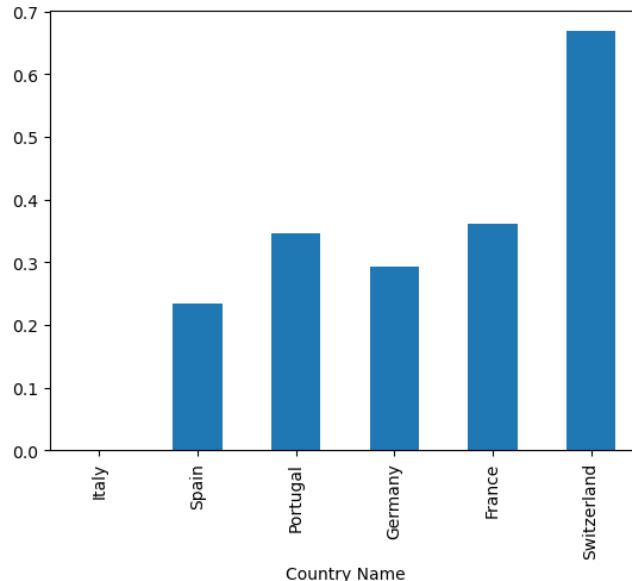


Un lineplot però non è adatto per questo tipo di dato, perché visualizza un andamento che non esiste, trattandosi di dati relativi allo stesso momento ma in posizioni geografiche differenti

Visualizzazioni con Pandas

Per cambiare il tipo di grafico, possiamo usare il parametro `kind=` del metodo `.plot()` per ottenere un **barplot** (grafico a barre), che è più appropriato:

```
pop.loc["1995"].plot(kind="bar")
```



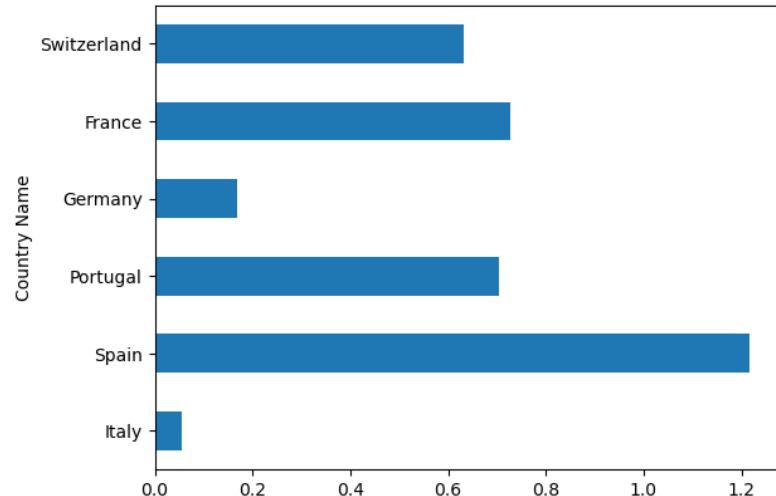
Visualizzazioni con Pandas

Per conoscere tutti i tipi di grafico ottenibili con il parametro `kind=` possiamo riferirci alla documentazione ufficiale di Pandas del metodo `.plot()`, e investigare la sezione relativa:

<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.html>

Ad esempio, utilizzando il valore "barh" possiamo ottenere un barplot orizzontale ("horizontal")

```
pop.loc["2001"].plot(kind="barh")
```



Visualizzazioni con Pandas

Un grafico molto importante per analizzare i dati, in particolare in dataset molto grandi, è l'**istogramma**, chiamato anche histplot.

Ad esempio, nella tabella *vw_factresellersales_denorm*, nella colonna *UnitPrice*, potremmo porre domande quali:

- Come si distribuiscono i dati?
- Qual è la fascia di prezzo che appare più spesso?
- Ci sono molti prezzi compresi tra 1000 e 1500?
- E tra 1500 e 2000?

Visualizzazioni con Pandas

Potremmo utilizzare `.value_counts()`:

```
df.UnitPrice.value_counts()

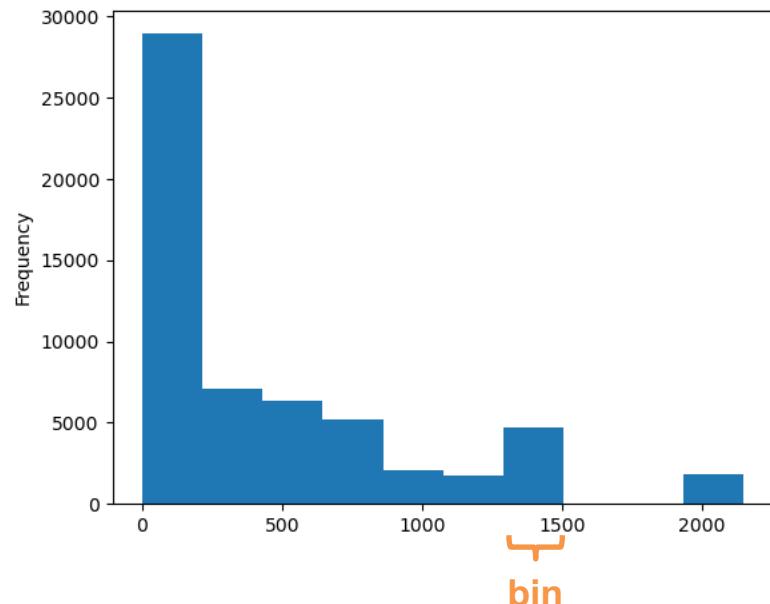
469.79      2969
419.46      2198
28.84       1593
20.19       1493
323.99      1463
...
313.64       1
2.50         1
327.16       1
791.41       1
15.73       1
Name: UnitPrice, Length: 230, dtype: int64
```

...ma in questo caso non ci dice molto.

Visualizzazioni con Pandas

Un istogramma in questo caso è molto più esplicativo:

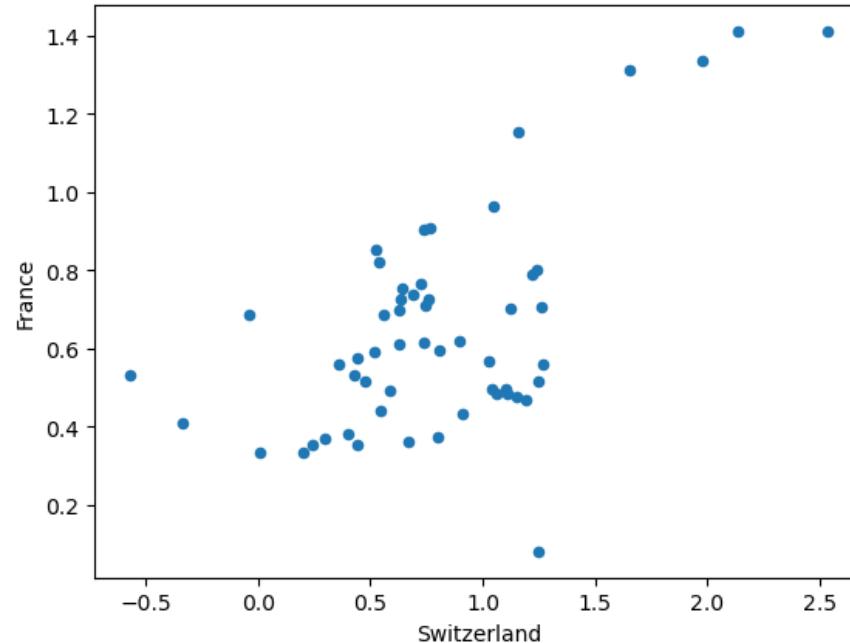
```
df.UnitPrice.plot(kind="hist")
```



Visualizzazioni con Pandas

Un grafico che permette di comparare i valori di due colonne numeriche è lo **scatterplot**:

```
pop.plot(kind="scatter", x="Switzerland", y="France")
```



Visualizzazioni con Seaborn

Per visualizzare i dati con dei **grafici** possiamo appoggiarci anche ad altre librerie Python.

Una di queste è **Seaborn**: <https://seaborn.pydata.org/>

Seaborn è progettato per funzionare bene con Pandas, permettendoci di creare grafici usando come base un DataFrame ed ottenendo delle visualizzazioni che, già di default, hanno un aspetto grafico curato e professionale, ma che possono essere ulteriormente personalizzate in modo intuitivo.

Seaborn, così come il metodo `.plot()` dei DataFrame, si basano sulla libreria Matplotlib ("MatLab Plotting Library", <https://matplotlib.org/>) tramite la quale si possono ottenere opzioni di personalizzazione più avanzate.

Molti dei dataset che vedremo possono essere scaricati qui:

<https://github.com/mwaskom/seaborn-data>

Visualizzazioni con Seaborn

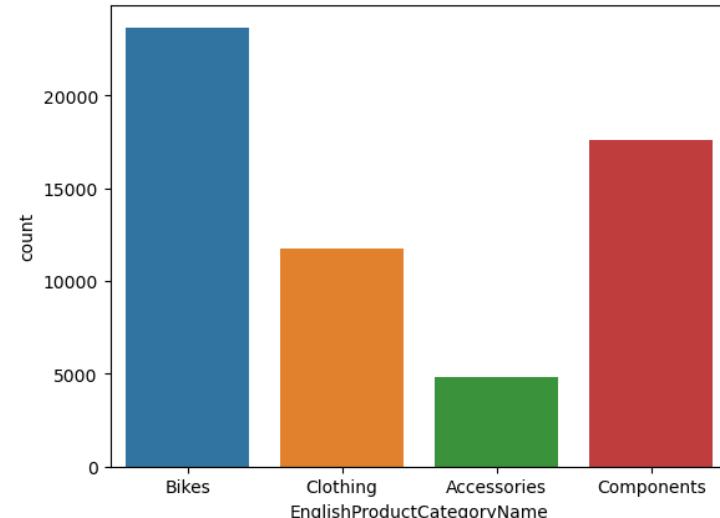
Per visualizzare la distribuzione delle colonne numeriche, possiamo usare diverse tipologie di grafico:

- **Countplot**: conta le istanze univoche di una colonna e ne realizza un barplot; in pratica la versione grafica di `.value_counts()`
- **Boxplot**: mostra la distribuzione di una variabile numerica attraverso i suoi quartili, evidenziando eventuali valori anomali
- **Jointplot**: mostra un grafico congiunto su due colonne, mostrando uno scatterplot unito agli istogrammi; si possono aggiungere altri elementi, quali una linea di regressione
- **Heatmap**: mostra la comparazione di dati numerici rispetto a dati categorici sottoforma di barre, che sono facilmente comparabili tra loro

Visualizzazioni con Seaborn

Visualizziamo quante tipologie di prodotto esistono nella tabella vw_factresellersales_denorm:

```
import seaborn as sns  
sns.countplot(data=df, x="EnglishProductName")
```



Vediamo che Seaborn è pensato per lavorare con i DataFrame e le relative colonne.

Visualizzazioni con Seaborn

Leggiamo il dataset `healthexp` con l'aspettativa di vita in diversi paesi per diversi anni:

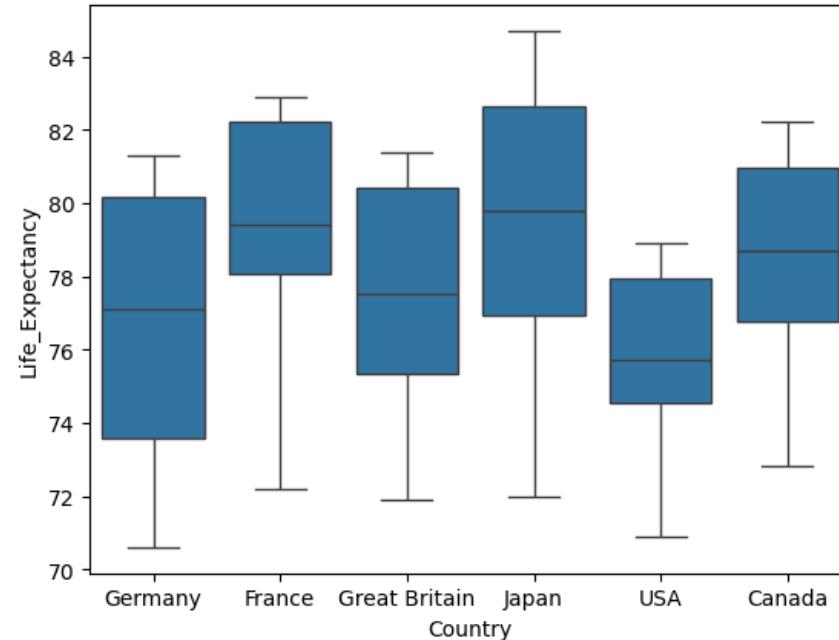
```
url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/refs/heads/master/healthexp.csv"
healthexp = pd.read_csv(url)
```

	Year	Country	Spending_USD	Life_Expectancy
0	1970	Germany	252.311	70.6
1	1970	France	192.143	72.2
2	1970	Great Britain	123.993	71.9
3	1970	Japan	150.437	72.0
4	1970	USA	326.961	70.9
...
269	2020	Germany	6938.983	81.1
270	2020	France	5468.418	82.3
271	2020	Great Britain	5018.700	80.4
272	2020	Japan	4665.641	84.7
273	2020	USA	11859.179	77.0

Visualizzazioni con Seaborn

Realizziamo un **boxplot** su questo dataset; per realizzare un boxplot c'è sempre bisogno di una colonna categorica e di una colonna numerica:

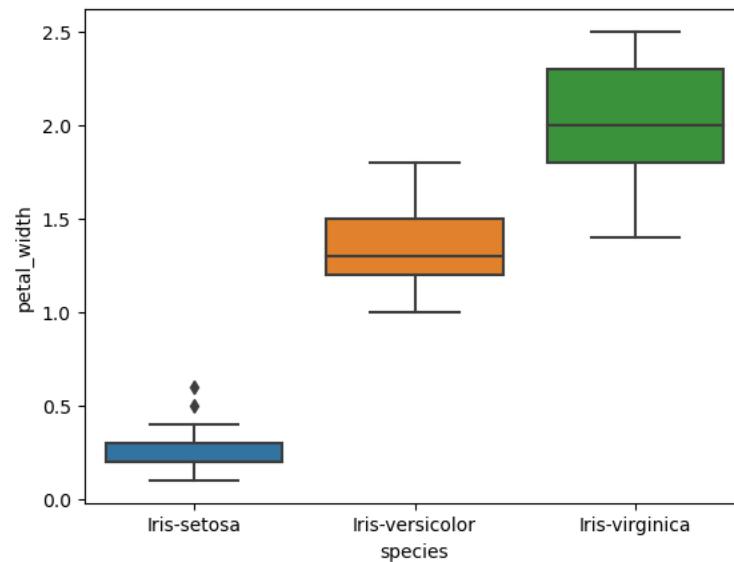
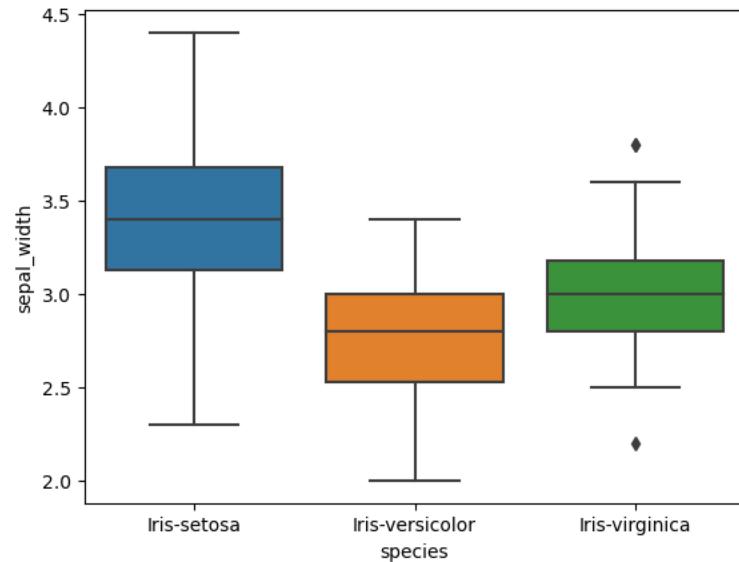
```
sns.boxplot(data=healthexp, x="Country", y="Life_Expectancy")
```



Visualizzazioni con Seaborn

Dai beginner_datasets su `iris.csv` vediamo le differenze tra le diverse specie con un boxplot:

```
iris = pd.read_csv("<PATH>/beginner_datasets/iris.csv")
sns.boxplot(data=iris, x="species", y="sepal_width")
sns.boxplot(data=iris, x="species", y="petal_width")
```



Visualizzazioni con Seaborn

Leggiamo il dataset mpg:

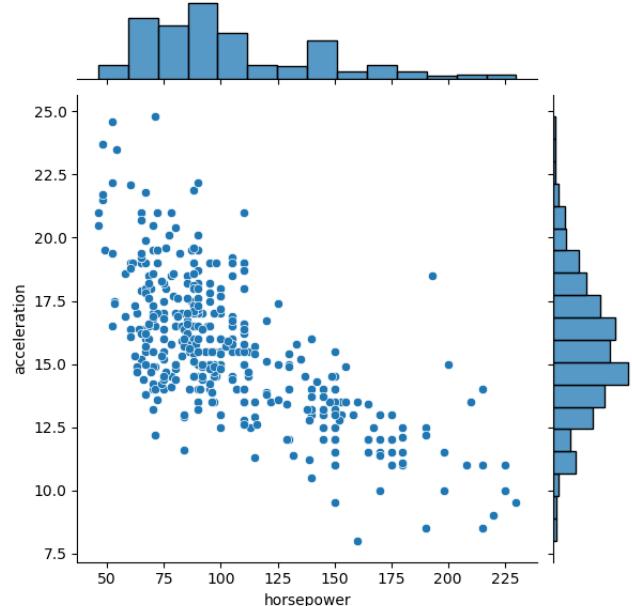
```
url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/refs/heads/master/mpg.csv"
auto = pd.read_csv(url)
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name	
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chevelle malibu	
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylark 320	
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth satellite	
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel sst	
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino	
...	
393	27.0	4	140.0	86.0	2790	15.6	82	usa	ford mustang gl	
394	44.0	4	97.0	52.0	2130	24.6	82	europe	vw pickup	
395	32.0	4	135.0	84.0	2295	11.6	82	usa	dodge rampage	
396	28.0	4	120.0	79.0	2625	18.6	82	usa	ford ranger	
397	31.0	4	119.0	82.0	2720	19.4	82	usa	chevy s-10	

Visualizzazioni con Seaborn

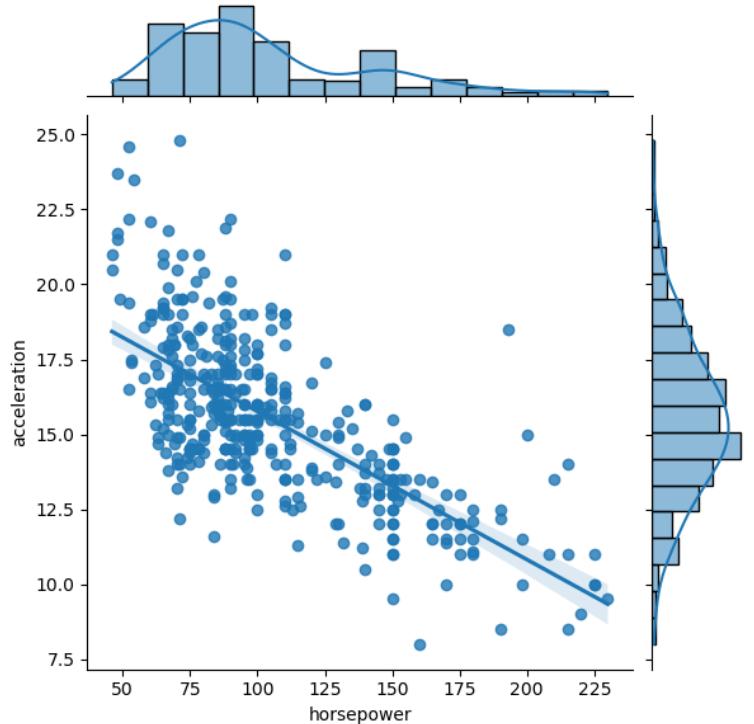
Per comparare due colonne possiamo usare la funzione `sns.jointplot()`:

```
url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/refs/heads/master/mpg.csv"
auto = pd.read_csv(url)
sns.jointplot(data=auto, x="horsepower", y="acceleration")
```



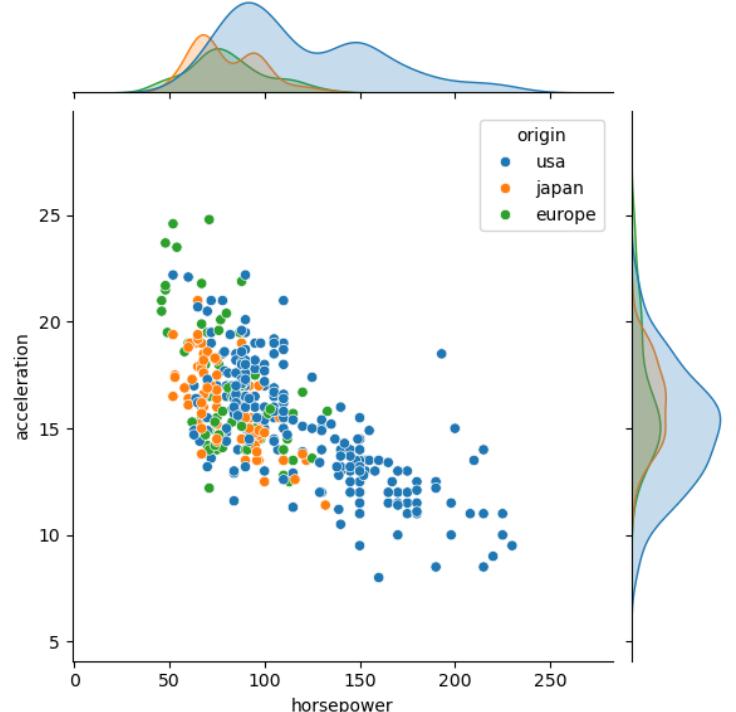
Visualizzazioni con Seaborn

```
sns.jointplot(data=auto, x="horsepower", y="acceleration", kind="reg")
```



Visualizzazioni con Seaborn

```
sns.jointplot(data=auto, x="horsepower", y="acceleration", hue="origin")
```



Visualizzazioni con Seaborn

Leggiamo il dataset `flights` che contiene i dati del numero di passeggeri di una tratta aerea per ogni mese dal 1949 al 1960:

```
url = "https://raw.githubusercontent.com/mwaskom/seaborn-  
data/refs/heads/master/flights.csv"  
flights = pd.read_csv(url)
```

	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121
...
139	1960	August	606
140	1960	September	508
141	1960	October	461
142	1960	November	390
143	1960	December	432

Visualizzazioni con Seaborn

È possibile riorganizzare i dati di un DataFrame mediante un'operazione di pivoting, mediante il metodo `.pivot_table()`

I parametri di questo metodo richiedono tre colonne, una da trasformare in indice, una da trasformare in colonne, e una contenente i dati; riordiniamo le colonne con l'ordine giusto dei mesi tramite `.loc[]` (che possiamo utilizzare per selezionare, ma anche per riordinare le colonne)

```
pfl = flights.pivot_table(index="year", columns="month", values="passengers")  
  
column_order = ["January", "February", "March", "April", "May", "June",  
"July", "August", "September", "October", "November", "December"]  
  
pfl = pfl.loc[:, column_order]  
  
print(pfl)
```

Visualizzazioni con Seaborn

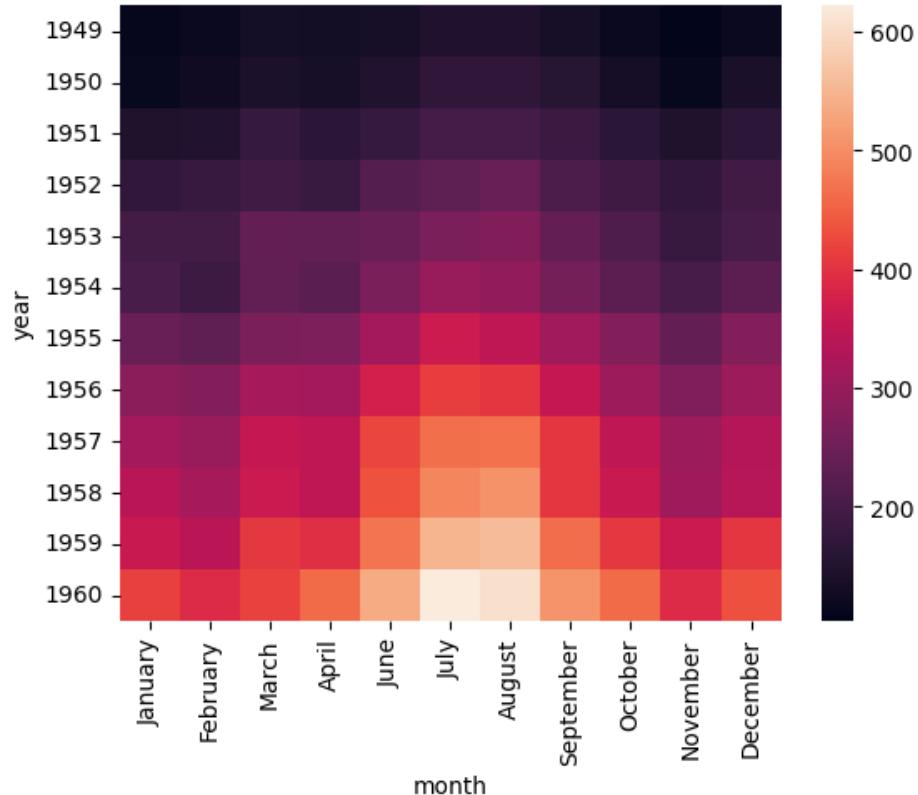
month	January	February	March	April	June	July	August	September	October	November	December
year											
1949	112.0	118.0	132.0	129.0	135.0	148.0	148.0	136.0	119.0	104.0	118.0
1950	115.0	126.0	141.0	135.0	149.0	170.0	170.0	158.0	133.0	114.0	140.0
1951	145.0	150.0	178.0	163.0	178.0	199.0	199.0	184.0	162.0	146.0	166.0
1952	171.0	180.0	193.0	181.0	218.0	230.0	242.0	209.0	191.0	172.0	194.0
1953	196.0	196.0	236.0	235.0	243.0	264.0	272.0	237.0	211.0	180.0	201.0
1954	204.0	188.0	235.0	227.0	264.0	302.0	293.0	259.0	229.0	203.0	229.0
1955	242.0	233.0	267.0	269.0	315.0	364.0	347.0	312.0	274.0	237.0	278.0
1956	284.0	277.0	317.0	313.0	374.0	413.0	405.0	355.0	306.0	271.0	306.0
1957	315.0	301.0	356.0	348.0	422.0	465.0	467.0	404.0	347.0	305.0	336.0
1958	340.0	318.0	362.0	348.0	435.0	491.0	505.0	404.0	359.0	310.0	337.0
1959	360.0	342.0	406.0	396.0	472.0	548.0	559.0	463.0	407.0	362.0	405.0
1960	417.0	391.0	419.0	461.0	535.0	622.0	606.0	508.0	461.0	390.0	432.0

Visualizzazioni con Seaborn

Un altro grafico che può risultare interessante è una **heatmap**, che visualizza la grandezza (potremmo dire, intuitivamente, l'"intensità") dei valori tramite colori. Ha senso solo in relazione alle variabili numeriche.

```
sns.heatmap(pfl)
```

Visualizzazioni con Seaborn



Visualizzazioni

Esistono molti altri tipi di grafico, sia legati a **concetti statistici più avanzati**, sia legati a **tipologie particolari di dimensioni** (ad esempio, grafici time-series per gli intervalli temporali, o mappe per i dati geografici).

Abbiamo visto insieme quelli fondamentali: a seconda delle necessità del momento possiamo andare a cercarne di più puntuali e imparare quali sono i parametri necessari a visualizzare il dato che ci interessa.

Il posto migliore dove iniziare è il Python Graph Gallery (<https://python-graph-gallery.com/>), che suddivide i tipi di grafico per tipologia e dà diversi esempi per ognuno con tanto di codice Python.





GRAZIE
EPCODE