



Corso di Apprendistato Python

Mod. Python
Docente:
Tonino Petrulli



Introduzione alla libreria NumPy

Imparare NumPy in poco tempo è una sfida. Non serve studiare ogni singola funzione, ma capire come ragiona la libreria.

Ecco il percorso ottimale suddiviso per importanza tecnica:

1. L'Anatomia dell'Array (ndarray)

Il punto di partenza è capire che un array NumPy non è una lista Python. È un blocco di memoria contiguo dove tutti gli elementi hanno lo stesso tipo.

Creazione: `np.array()`, `np.zeros()`, `np.ones()`, `np.linspace()`.

Attributi fondamentali: `.shape` (la dimensione), `.ndim` (il numero di assi) e `.dtype` (il tipo di dati, fondamentale per la memoria).

2. Slicing e Indexing (Accesso ai dati)

In un'ora devi padroneggiare come estrarre righe, colonne o sotto-matrici.

Slicing 2D: Capire la sintassi `array[righe, colonne]`. Ad esempio, `arr[:, 1]` per prendere tutta la seconda colonna.

Boolean Indexing: Filtrare i dati senza usare `if`. Es: `prezzi[prezzi > 100]` ti restituisce solo i valori sopra i cento. È la funzione più potente per l'analisi dati veloce.

Introduzione alla libreria NumPy

3. Vettorizzazione e Operazioni Element-wise

Questa è la ragione per cui NumPy esiste. Invece di iterare sugli elementi, applichi l'operazione all'intero oggetto.

Se fai `array * 2`, NumPy moltiplica ogni singolo elemento per due istantaneamente a livello C (molto più veloce di un ciclo Python).

Funzioni Universali (ufuncs): `np.sin()`, `np.exp()`, `np.log()`.

4. Il concetto di Broadcasting

Questo è l'argomento più "difficile" ma essenziale. Spiega come NumPy gestisce operazioni tra array di dimensioni diverse.

Esempio: Sommare un vettore riga a ogni riga di una matrice. Comprendere le regole di compatibilità delle forme (shape) ti eviterà ore di errori criptici.

5. Aggregazioni e Assi (axis)

Devi sapere come riassumere i dati.

Funzioni come `.sum()`, `.mean()`, `.max()`.

Il segreto degli assi: Capire la differenza tra `axis=0` (operazioni lungo le colonne) e `axis=1` (operazioni lungo le righe). Se sbagli l'asse, il tuo calcolo scientifico sarà logicamente errato.

La scienza dei dati (Data science)

1 Introduzione alla scienza dei dati

La **scienza dei dati**, più comunemente chiamata **Data Science**, è la disciplina che si occupa dello studio dei dati. L'obiettivo primario di questa scienza è l'**interpretazione** e l'**estrazione di informazioni complesse** a partire dai dati raccolti.

Per esempio, dallo storico degli ordini di un negozio è possibile estrarre un elenco degli articoli più venduti e sapere quelli che hanno un maggior margine di profitto, oppure, al contrario, quelli che sono rimasti invenduti e occupano spazio prezioso nel magazzino. Colui che si occupa di svolgere queste analisi è chiamato **scienziato o scienziata dei dati**.

Questa figura è quella di un vero e proprio esperto dei dati in tutti i loro aspetti: sa come i dati sono stati raccolti, conosce il loro significato, sceglie la rappresentazione più adatta e decide le operazioni necessarie per estrarre le informazioni richieste. Ma perché parliamo di «scienza» dei dati e non «analisi» dei dati? Perché «scienziato» dei dati e non «analista» dei dati? La risposta è semplice: a differenza dell'analisi dei dati, nella quale ci si limita a estrarre informazione dai dati, nella scienza dei dati estraiamo informazione con lo scopo ultimo di confermare o meno un'**ipotesi** iniziale. In sostanza, applichiamo il metodo scientifico al campo dell'informazione.

La scienza dei dati (Data science)

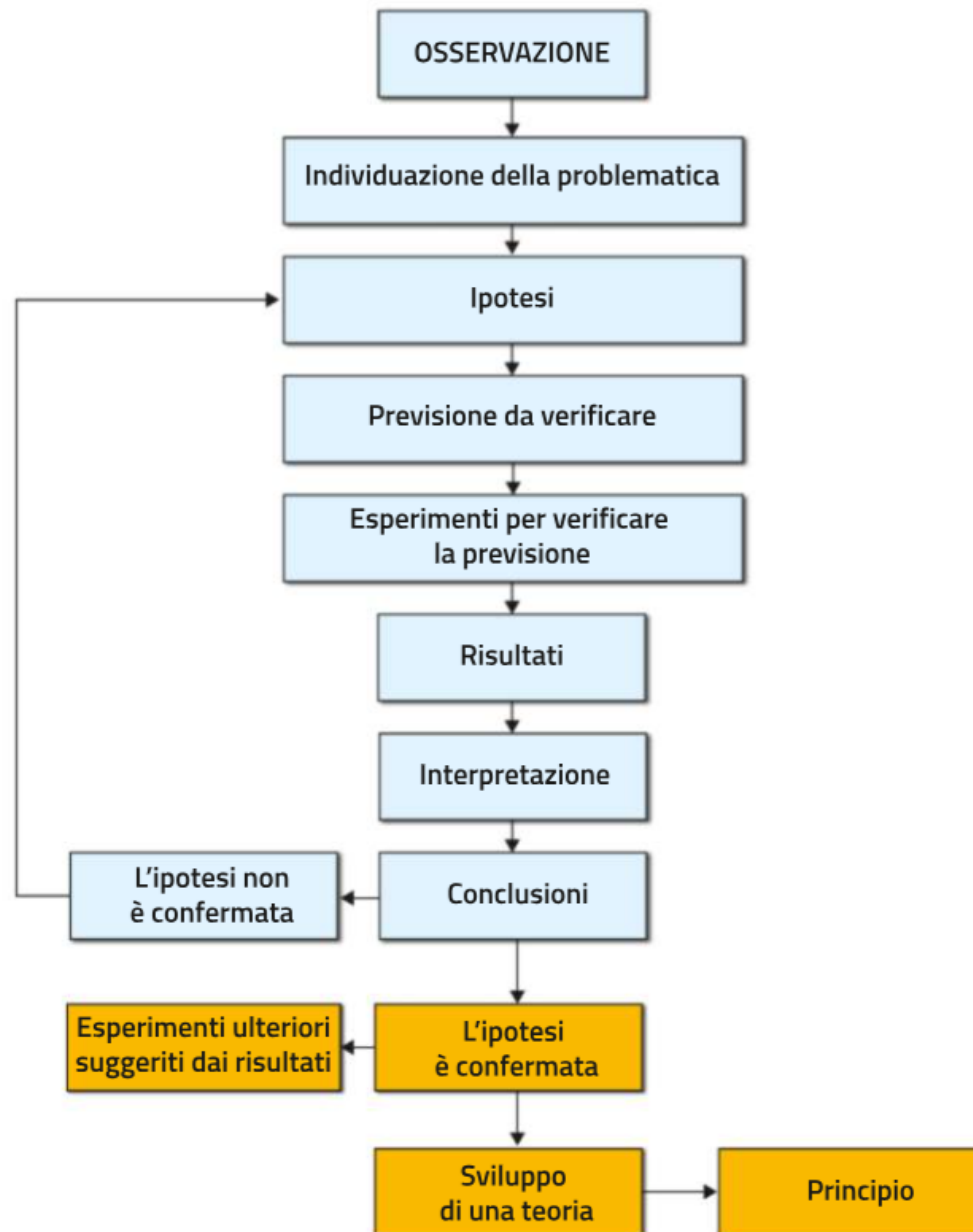
Il **metodo scientifico** è un processo per cui uno scienziato formula un'ipotesi e mette in atto una serie di passaggi per confermare o smentire tale ipotesi. Esistono diversi modi di procedere per raggiungere questo scopo. Uno di questi è il **metodo induttivo**, che si compone delle seguenti fasi.

1. **Osservazione.** Costituisce il primo passo del metodo induttivo, e consiste nel raccogliere informazioni sul fenomeno che stiamo studiando.
2. **Formulazione di un'ipotesi.** Dall'osservazione del fenomeno è possibile derivare una possibile spiegazione (un'ipotesi) di quello che stiamo osservando.
3. **Esecuzione di un esperimento.** L'ipotesi deve essere messa alla prova, in modo da poter essere certi che sia una spiegazione coerente con quello che abbiamo osservato. Per fare ciò bisogna eseguire un esperimento, interagendo con il fenomeno e raccogliendo delle nuove osservazioni.
4. **Interpretazione dei risultati.** Dalle informazioni che abbiamo raccolto cerchiamo di capire se l'ipotesi che abbiamo formulato è confermata oppure no.
5. **Conclusione.** Al termine dell'interpretazione dei risultati è possibile decidere se l'ipotesi è stata confermata oppure no.

Se le conclusioni tratte dall'interpretazione dei risultati sperimentali sono in accordo con l'ipotesi formulata, allora abbiamo confermato l'ipotesi iniziale; altrimenti, è necessario ripetere il processo dal punto 2, formulando una nuova ipotesi. **Fig. 1** rappresenta il metodo induttivo tramite un diagramma di flusso, illustrando anche altri passaggi intermedi che ci aiutano a comprendere meglio l'intero percorso.

La scienza dei dati (Data science)

◀ Fig. 1 Il metodo induttivo.

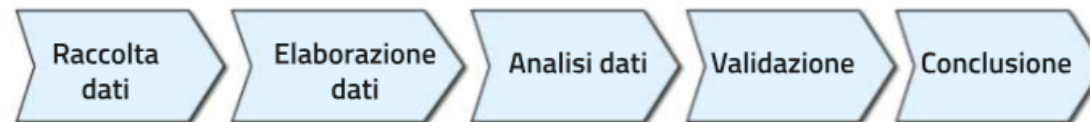


La scienza dei dati (Data science)

Il metodo induttivo viene applicato in molte discipline, come la Chimica, la Fisica e la Biologia. La differenza tra queste scienze e l'Informatica sta nella fase sperimentale: mentre negli altri casi condurre un esperimento significa spesso interagire con la realtà per dati grezzi, per esempio una reazione chimica, nel caso della Data Science l'**analisi dei dati costituisce la fase sperimentale**.

Per **dati grezzi** intendiamo quei dati che sono stati raccolti ma non sono ancora stati elaborati, analizzati o trasformati in alcun modo. I dati grezzi rappresentano la forma più elementare delle informazioni raccolte e non sono stati sottoposti a nessun processo di pulizia, normalizzazione, aggregazione o interpretazione.

Questi passaggi costituiscono quella che viene chiamata **Data Science pipeline** (o Data Science *workflow*), ossia la sequenza di passaggi che portano dai dati grezzi alla conclusione finale, come mostrato in Fig. 2.



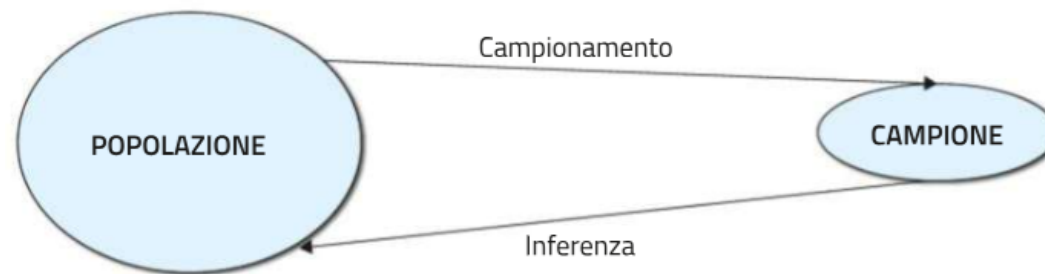
Per la precisione, Fig. 2 rappresenta un esempio di *pipeline* generica: a seconda della fase di analisi che dobbiamo effettuare, la *pipeline* può includere passaggi specifici per quel caso particolare. Mentre raccolta dati e conclusione dipendono strettamente dal problema che bisogna risolvere, elaborazione, analisi e validazione condividono caratteristiche comuni in tutte le *pipeline*.

In questo Capitolo trattiamo brevemente della raccolta dei dati e ci concentriamo maggiormente sugli strumenti che permettono di manipolare i dati, mentre nel Capitolo successivo entreremo nel dettaglio di elaborazione, analisi e validazione tramite tecniche di Intelligenza Artificiale.

La scienza dei dati (Data science)

1.1 Campionamento e inferenza

Un tema fondamentale nella Data Science è la **rappresentazione dell'informazione**. Nella programmazione a oggetti abbiamo visto che una classe è un'astrazione della realtà che contiene solo gli attributi e i metodi che servono al corretto funzionamento del programma; si basa quindi su una **semplificazione della realtà**. Allo stesso modo, nella Data Science, i dati raccolti forniscono un'**osservazione parziale** di quello che succede nel mondo reale. Per comprendere meglio questo aspetto è necessario introdurre i concetti di **popolazione** e **campione** [Fig. 3].



Per **popolazione** intendiamo l'insieme delle istanze, anche dette unità, che sono oggetto di studio. Un **campione** è un sottoinsieme delle unità contenute nella popolazione.

Se facciamo l'esempio di uno studio sulle preferenze culinarie locali, la popolazione di interesse è quella di una città, mentre il campione è dato da un numero ristretto di cittadini. Non è possibile raccogliere dati da tutti i cittadini: potrebbe non essere fattibile, sia in termini di costi sia di tempo. Per questo motivo si sceglie di **campionare**, cioè selezionare, dei cittadini che rappresentano l'intera popolazione. Il campione viene poi analizzato per estrarre informazioni che descrivono il comportamento dell'intera popolazione. Questo processo viene chiamato **inferenza**.

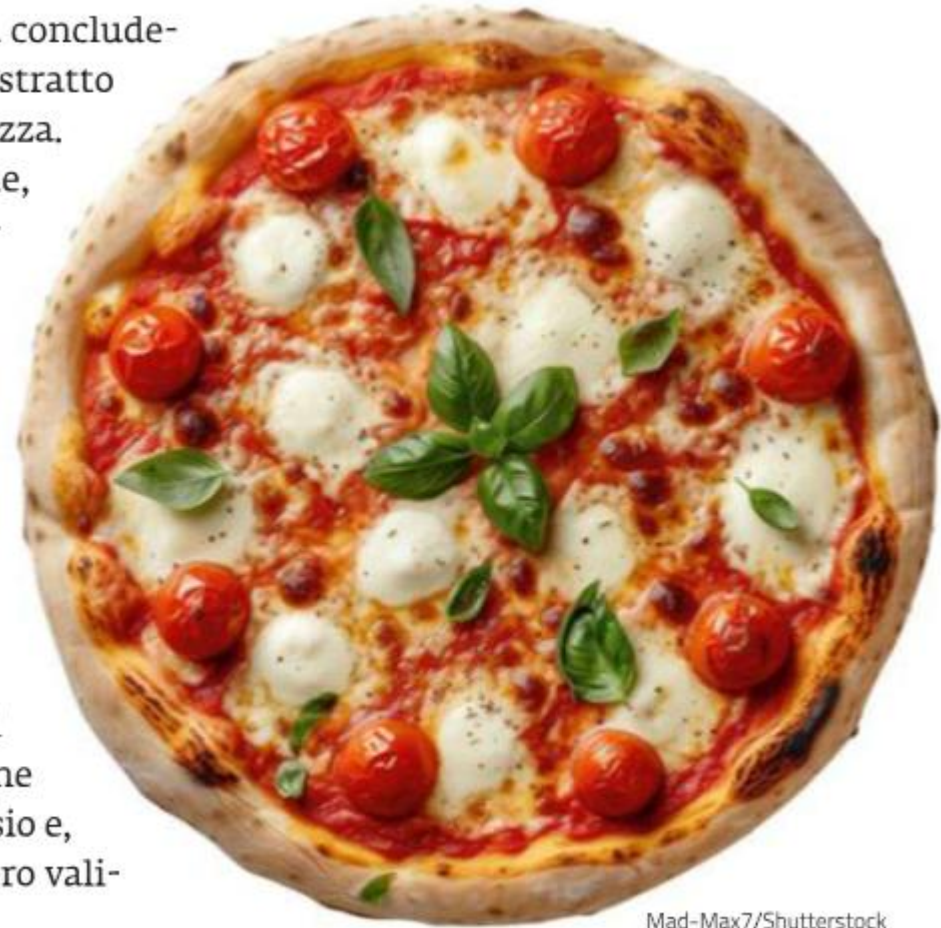
Per **inferenza** intendiamo la generalizzazione dei risultati ottenuti dall'analisi di un campione all'intera popolazione da cui è stato estratto il campione stesso.

La scienza dei dati (Data science)

In pratica, se dall'analisi del campione che abbiamo collezionato osserviamo che il piatto preferito è la pizza, allora concluderemo che in quella popolazione da cui abbiamo estratto il campione il piatto preferito è per l'appunto la pizza.

Anche se questo ragionamento sembra banale, non è detto che ciò che osserviamo in un campione sia vero anche nella popolazione generale. Infatti, perché l'inferenza sia generalizzabile alla popolazione, il campione deve essere **rappresentativo** della popolazione, ossia, deve avere caratteristiche simili alla popolazione da cui è stato estratto.

Se, per esempio, nel campione selezionassimo solo individui intolleranti al lattosio, escluderemmo automaticamente tutti i piatti che contengono derivati del latte. In questo modo, il campione non rappresenterebbe più la popolazione generale, ma solo quella parte intollerante al lattosio e, di conseguenza, i risultati dell'analisi non sarebbero validi nella popolazione generale.



Mad-Max7/Shutterstock

La scienza dei dati (Data science)

Abbiamo parlato di popolazione e campione a livello generale, ma l'analisi dei dati si basa su operazioni che dipendono dal **tipo di dati** che bisogna analizzare. Continuando l'analogia con la programmazione a oggetti, ogni unità del campione ha delle caratteristiche, cioè degli attributi, che chiamiamo **variabili**. Per esempio, una persona ha un'età, un genere e un colore preferito. Le variabili possono essere divise in due categorie:

- **variabili quantitative**, che sono rappresentate da **valori numerici**, per esempio il peso o l'età di una persona,
- **variabili qualitative**, che sono rappresentate da **valori non numerici**, per esempio il genere di una persona o il suo colore preferito.

Come vedremo nelle prossime pagine, conoscere il tipo di una variabile ci permette di sapere quali sono le informazioni che possiamo estrarre da un campione.

La scienza dei dati (Data science)

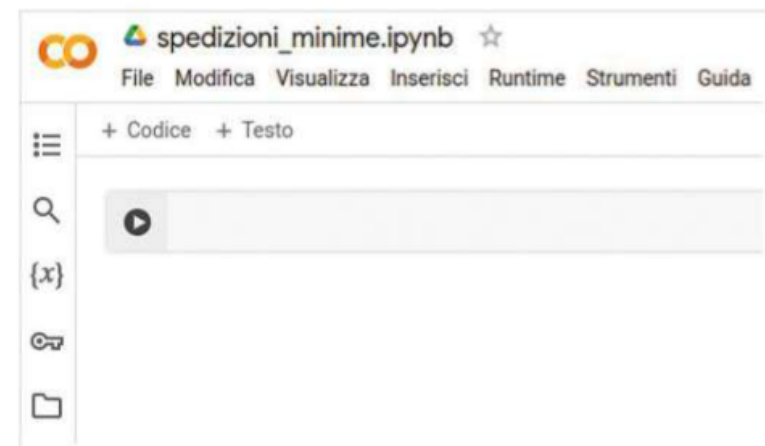
1.2 Colaboratory

Finora abbiamo utilizzato l'IDLE di Python per eseguire codice e visualizzare l'output. Questo strumento è fondamentale per l'esecuzione di pezzi di codice isolati, ma in Data Science si tende a scrivere delle *pipeline* e a commentare dei blocchi di codice in maniera estesa. Inoltre, spesso più persone collaborano sullo stesso progetto, e avere un'interfaccia grafica più facile da utilizzare dell'IDLE rende la collaborazione più facile. Per questi motivi, faremo uso di un nuovo ambiente, realizzato da Google e pensato proprio per l'analisi dei dati: **Colaboratory**. Chiamato anche **Colab**, questo ambiente di sviluppo fornisce un'interfaccia web ed esegue il codice Python in remoto sui server di Google gratuitamente. Per capire come utilizzare questo ambiente consideriamo di nuovo l'Esempio 13 del Capitolo 5, per il quale ora useremo Colab.

Noteremo che Colab evidenzia gli elementi del codice Python con colori diversi da quelli dell'editor ufficiale che abbiamo utilizzato finora. Ciò non ci deve sorprendere: non è strano che editor diversi seguano regole di visualizzazione diverse, l'importante è che ciascuno al suo interno sia coerente.

Per prima cosa apriamo il browser e quindi accediamo a Colab, disponibile all'indirizzo <https://colab.research.google.com>. L'interfaccia di Colab si basa sul concetto di **notebook**, ovvero blocco note, in cui si alternano gruppi, chiamati **celle**, di righe di codice e di testo. Per creare un nuovo notebook basta aprire il menù a tendina *File* e cliccare su *Nuovo blocco note*. In alternativa, si può selezionare *Carica blocco note* e caricare il file collegato.

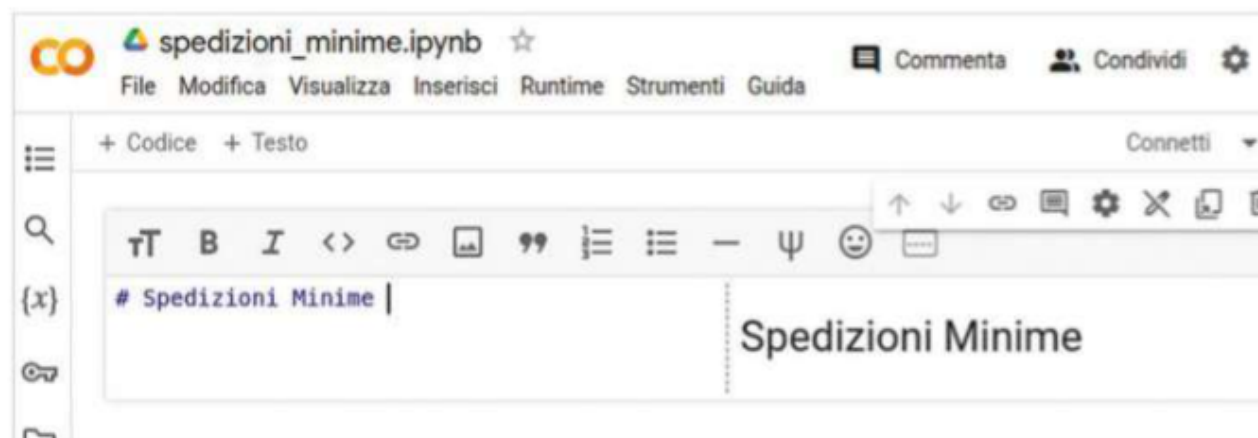
Si aprirà una schermata simile a quella di **Fig. 3**.



La scienza dei dati (Data science)

I due pulsanti *+ Codice* e *+ Testo* ci permettono di aggiungere rispettivamente codice in Python oppure testo in Markdown (<https://it.wikipedia.org/wiki/Markdown>), un semplice linguaggio di mark-up che permette di formattare il testo.

Per esempio, per aggiungere un titolo al notebook possiamo cliccare sul pulsante *+ Testo* e scrivere come titolo `# Spedizioni Minime`. In questo caso, il cancelletto non individua più un commento ma un titolo, e il numero di cancelletti determina il «livello», cioè la grandezza, del titolo: un cancelletto corrisponde al livello 1 e quindi alla dimensione più grande del testo, due cancelletti al livello 2 e quindi più piccolo del livello 1, e così via [Fig. 4].

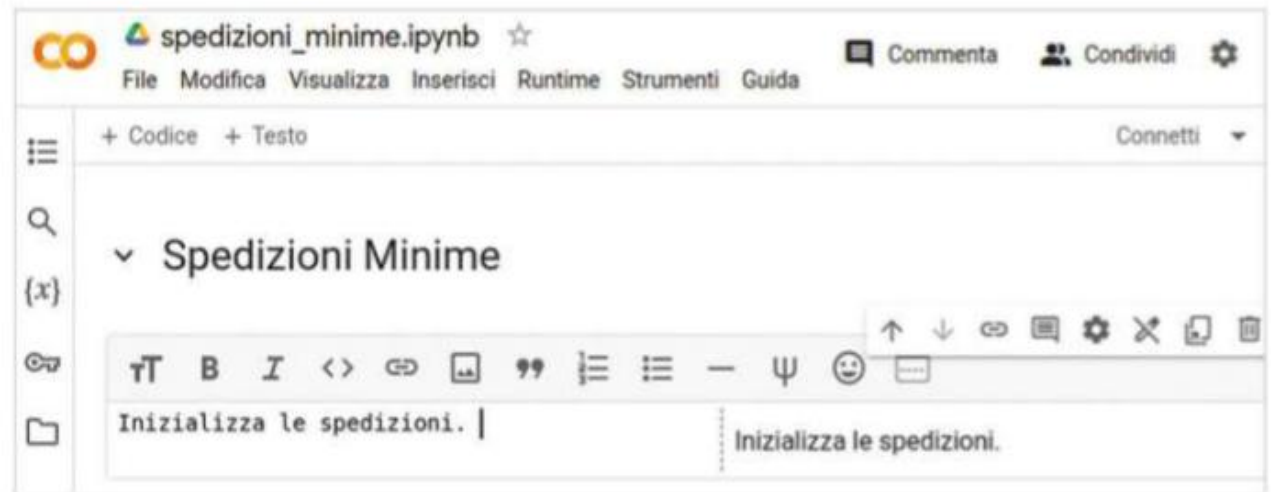


Colab è provvisto di un editor integrato nella cella di testo, quindi non è necessario conoscere il linguaggio Markdown. Inoltre, a destra mostra un'anteprima di come il testo verrà visualizzato.

Per completezza, la prima cella di codice presente di default alla creazione del notebook è stata rimossa cliccando sull'icona del cestino in alto a destra, che si apre cliccando su ogni cella.

La scienza dei dati (Data science)

Se aggiungiamo un'altra cella di testo possiamo riportare il primo commento dell'Esempio, questa volta senza il cancelletto [Fig. 5].



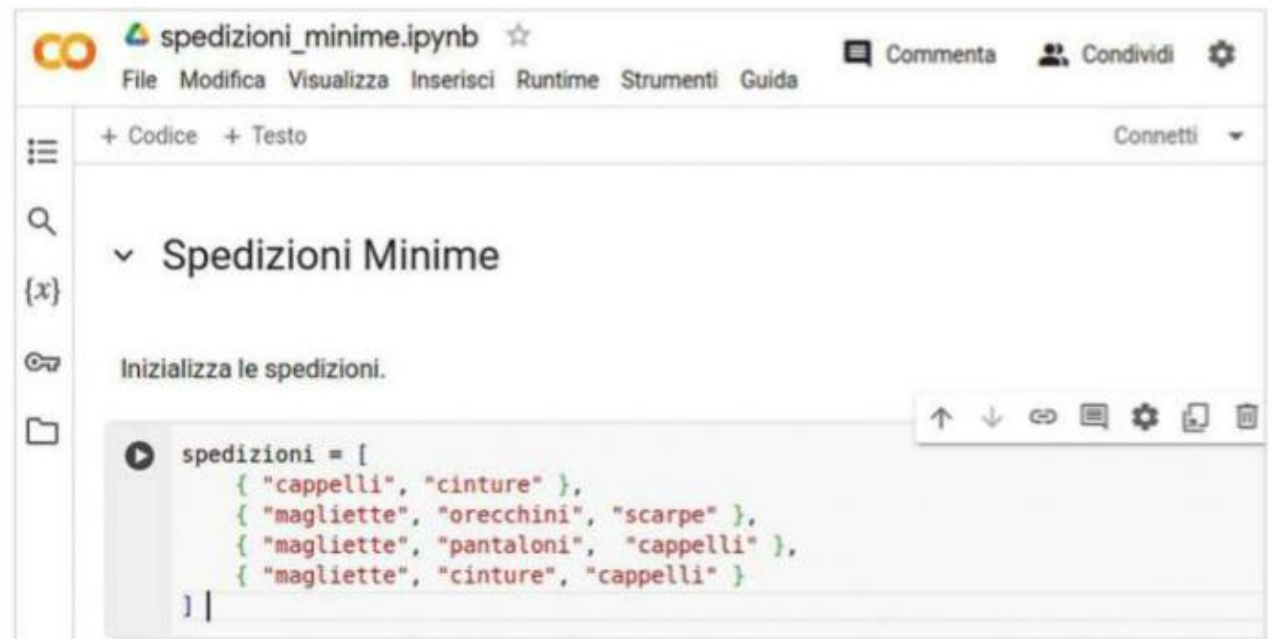
Allo stesso modo, aggiungiamo una cella cliccando sul pulsante + Codice [Fig. 6].

```
spedizioni = [  
    {"cappelli", "cinture"},  
    {"magliette", "orecchini", "scarpe"},  
    {"magliette", "pantaloni", "cappelli"},  
    {"magliette", "cinture", "cappelli"}  
]
```

N.B.: Questo codice definisce una variabile chiamata `spedizioni`, che è una **lista di set (insiemi)**.

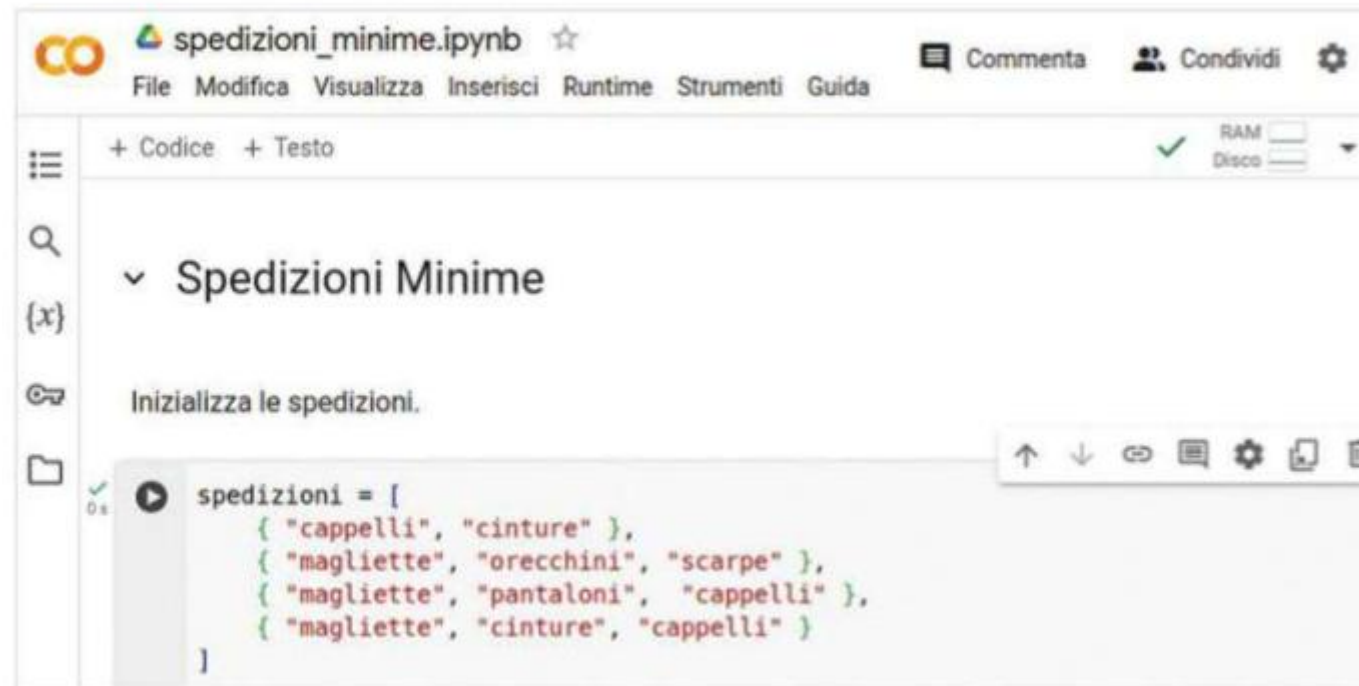
Ogni elemento della lista rappresenta un ordine o un pacco, e all'interno di ogni pacco ci sono diversi prodotti.

Questa struttura è perfetta per operazioni matematiche sugli insiemi.



La scienza dei dati (Data science)

Per eseguire il codice basta cliccare sul pulsante *Play*. La prima volta che viene eseguito del codice in un notebook, vengono allocate le risorse sul server remoto di Google [Fig. 7].



Come si può notare, in alto a destra sono comparsi due indicatori, **RAM** e **Disco**, che segnalano l'utilizzo corrente delle risorse. Inoltre, di fianco al pulsante *Play* vengono indicati lo stato e il tempo di esecuzione della riga di codice eseguita; in questo caso una spunta verde indica che l'esecuzione è andata a buon fine ed è durata 0 secondi, quindi è stata istantanea.

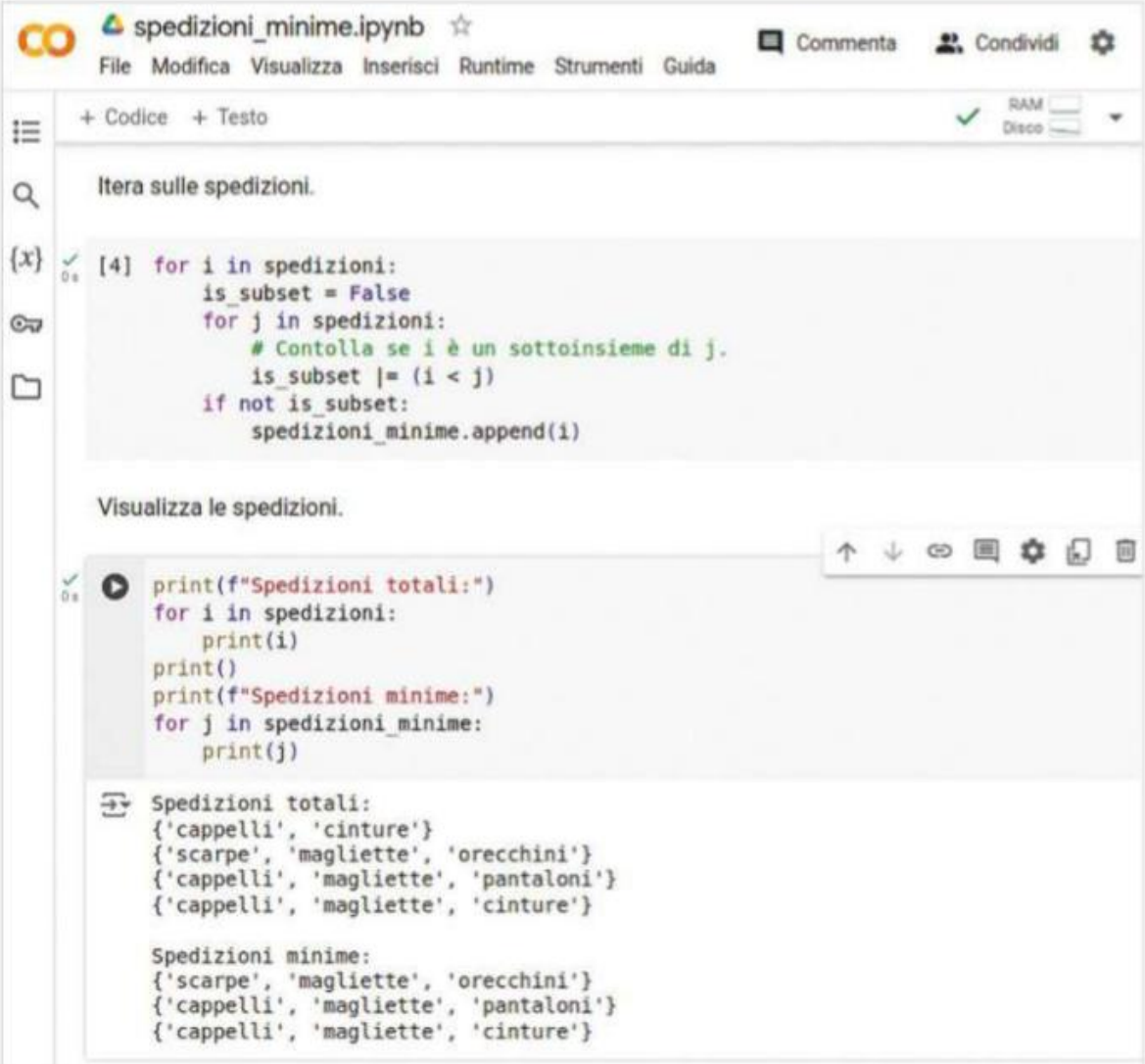
La scienza dei dati (Data science)

```
for i in spedizioni:
    is_subset = False
    for j in spedizioni:
        #Controlla se i è un sottoinsieme di j.
        is_subset |= (i < j)
    if not is_subset:
        spedizioni_minime.append(i)
```

```
print(f"Spedizioni totali:")
for i in spedizioni:
    print(i)
print()
print(f"Spedizioni minime:")
for j in spedizioni_minime:
    print(j)
```

C'è un errore! Ma... ci aiuta Gemini!

Possiamo continuare e riportare l'intero Esempio sul notebook, procedendo cella per cella, così da ottenere una sua versione interattiva e facilmente condivisibile [Fig. 8].



The screenshot shows a Jupyter Notebook interface with the title 'spedizioni_minime.ipynb'. The top bar includes a menu with 'File', 'Modifica', 'Visualizza', 'Inserisci', 'Runtime', 'Strumenti', and 'Guida'. On the right, there are buttons for 'Commenta', 'Condividi', and a settings icon. Below the menu, there are tabs for '+ Codice' and '+ Testo'. The first code cell is titled 'Itera sulle spedizioni.' and contains a nested loop that filters items from 'spedizioni' into 'spedizioni_minime' based on a subset condition. The second code cell is titled 'Visualizza le spedizioni.' and contains two print statements to display the total and minimal shipments. The output of the second cell shows two lists of dictionaries, each containing three items: 'cappelli', 'cinture', 'scarpe', 'magliette', 'orecchini', and 'pantaloni'.

```
[4] for i in spedizioni:
    is_subset = False
    for j in spedizioni:
        # Controlla se i è un sottoinsieme di j.
        is_subset |= (i < j)
    if not is_subset:
        spedizioni_minime.append(i)
```

```
print(f"Spedizioni totali:")
for i in spedizioni:
    print(i)
print()
print(f"Spedizioni minime:")
for j in spedizioni_minime:
    print(j)
```

Spedizioni totali:

```
{'cappelli', 'cinture'}
{'scarpe', 'magliette', 'orecchini'}
{'cappelli', 'magliette', 'pantaloni'}
{'cappelli', 'magliette', 'cinture'}
```

Spedizioni minime:

```
{'scarpe', 'magliette', 'orecchini'}
{'cappelli', 'magliette', 'pantaloni'}
{'cappelli', 'magliette', 'cinture'}
```

In questo modo è anche possibile isolare le varie parti della sequenza di lavoro e ragionare cella per cella, cioè blocchetto per blocchetto.

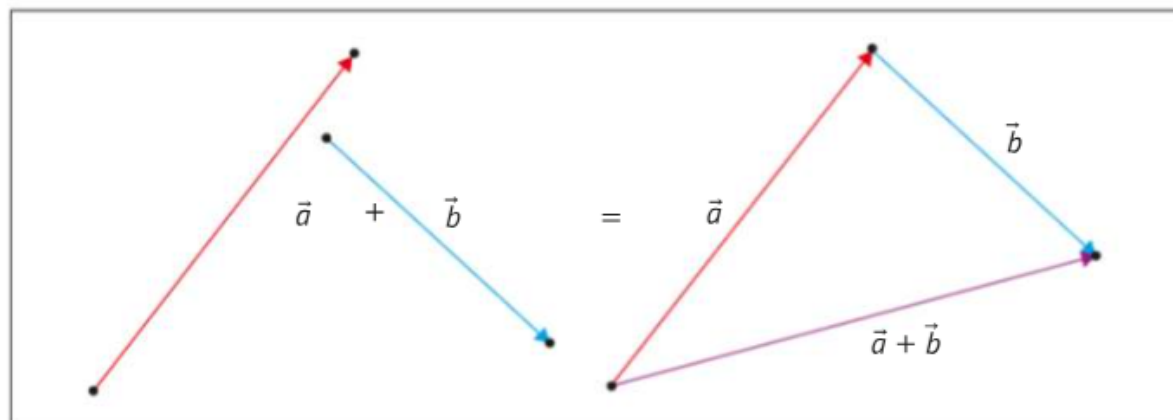
2 Operazioni su vettori e matrici con *Numpy*

2.1 I dati vettoriali

In Informatica un vettore (o array monodimensionale) è una sequenza di valori omogenei, ossia, dello stesso tipo. I vettori che useremo in questo Capitolo hanno delle funzionalità aggiuntive che espandono questa descrizione e li avvicinano alla **definizione Matematica** di vettore.

In Matematica, un **vettore** è un elemento di uno **spazio vettoriale**, ossia un insieme di elementi che supportano operazioni binarie, come addizione e moltiplicazione.

I valori che compongono un vettore sono detti **componenti del vettore**. È possibile rappresentare i vettori a due componenti su un piano cartesiano [Fig. 9].



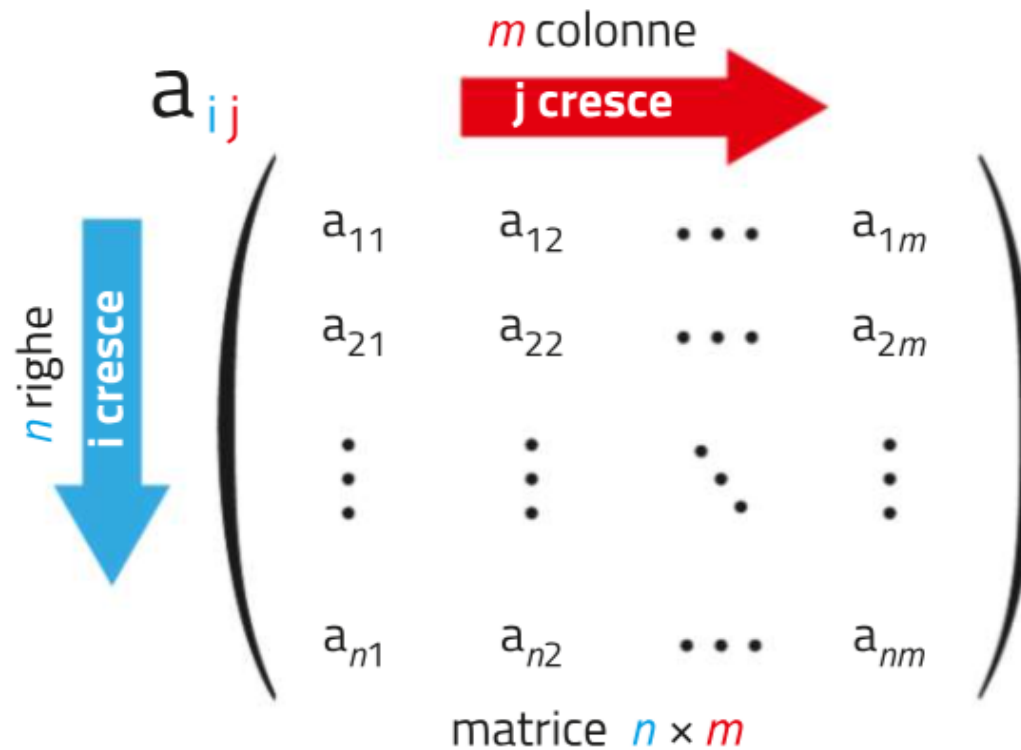
In Matematica un vettore supporta le operazioni elementari di somma, differenza, prodotto e quoziente. I vettori Python però non implementano questi metodi e, come vedremo nelle prossime pagine, è necessario utilizzare **pacchetti aggiuntivi** per sopperire a questa mancanza.

La libreria Numpy

Oltre ai vettori faremo uso delle matrici.

In Matematica, una **matrice** è una **tabella**, o array bidimensionale, di numeri ordinati in righe e colonne.

Possiamo rappresentare una matrice come in Fig. 10.



La libreria Numpy

2.2 Operazioni su vettori e matrici

Per rappresentare vettore, matrici e array n -dimensionali in Python si utilizza la libreria **Numpy**. Questa libreria è già installata nell'ambiente Colab, quindi è possibile importarla direttamente nella prima cella di codice del notebook.

```
✓ [1] import numpy as np
```

Dato che utilizzeremo molto questa libreria, decidiamo di abbreviare il nome alla sigla np, così da accorciare le righe di codice che scriveremo. La libreria offre molte funzionalità che possono essere utilizzate per gli scopi più disparati; qui ci concentreremo sulle funzionalità base, mentre per tutto il resto è possibile consultare la documentazione ufficiale (in inglese) all'indirizzo <https://numpy.org>.

La prima operazione fondamentale è la creazione di un vettore. Dato che *Numpy* non fa differenza tra vettori, matrici e array n -dimensionali, la funzione base per la creazione di un oggetto *Numpy* si chiama array.

```
✓ [2] data = np.array([1, 2, 3])
```

```
✓ [3] data
```

```
→ array([1, 2, 3])
```

La funzione accetta in input una lista di numeri e restituisce in output un array 1-dimensionale (in breve un array 1D) che supporta tutte le operazioni vettoriali matematiche più utilizzate.



La libreria Numpy

Per ognuna di queste funzionalità possiamo vedere graficamente come sono disposti gli elementi dell'array.

	data		ones				
<code>data = np.array([1,2])</code>	<table><tr><td>1</td></tr><tr><td>2</td></tr></table>	1	2	<code>ones = np.ones(2)</code>	<table><tr><td>1</td></tr><tr><td>1</td></tr></table>	1	1
1							
2							
1							
1							

Come per le liste in Python, gli array 1D di *Numpy* supportano l'accesso tramite indice (in inglese **indexing**), e la selezione di un sottosequenza dell'array, ovvero una «fetta» (in inglese **slicing**).

✓ 0s	[4] data[0]
	↔ 1
✓ 0s	[5] data[1]
	↔ 2
✓ 0s	[6] data[0:2]
	↔ array([1, 2])
✓ 0s	[7] data[1:]
	↔ array([2, 3])
✓ 0s	[8] data[-2:]
	↔ array([2, 3])

Mentre per accedere a una posizione dell'array indexing fa uso di un singolo indice numerico, slicing si basa sull'utilizzo di sequenze di valori, come il range 0:2, che identifica tutti gli indici da 0 (incluso) a 2 (escluso).

La libreria Numpy

Inoltre, sia indexing sia slicing supportano per gli indici valori negativi, che consentono di accedere agli elementi dell'array a partire dalla fine: -1 è l'indice dell'ultimo elemento dell'array, -2 del penultimo e così via, come per le liste e le stringhe che abbiamo visto nel Capitolo 5.

	data	data[0]	data[1]	data[0:2]	data[1:]	data[-2:]		data
0	1	1		1			0	1
1	2		2	2	2	2	1	2
2	3				3	3	2	3
							3	

La somma di due array si traduce immediatamente nella somma delle componenti che costituiscono gli array stessi. Il vantaggio di operare con array piuttosto che con liste è subito evidente: le operazioni elementari sono già implementate e non necessitano di essere re-implementate tramite cicli.

```
✓ [9] data, ones = np.array([1, 2]), np.array([1, 1])
```

```
✓ [10] data + ones
```

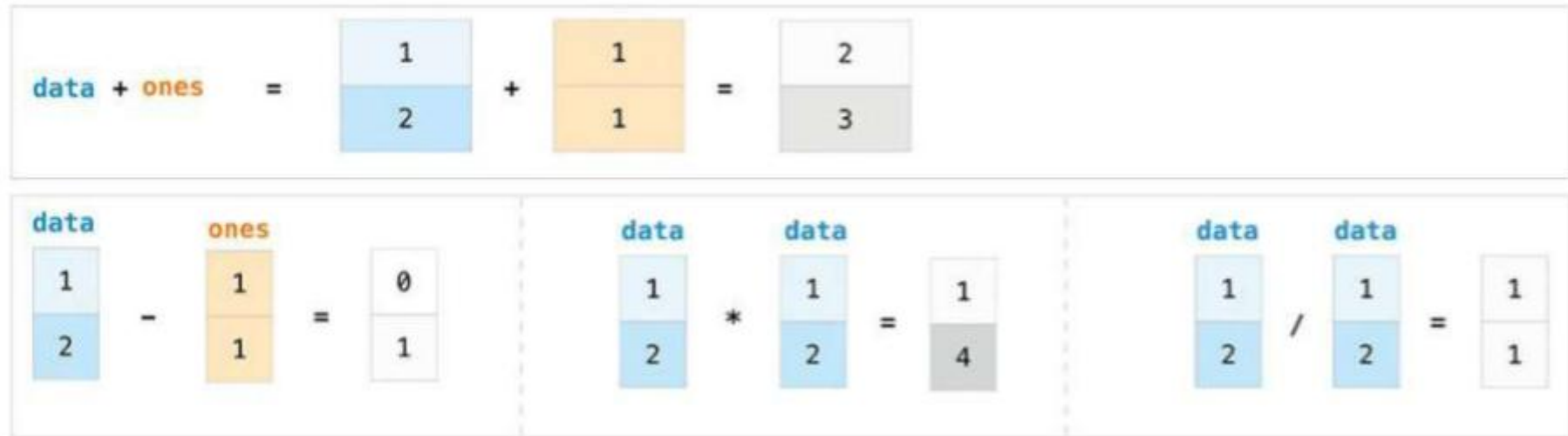
```
⇒ array([2, 3])
```

```
✓ # Differenza, prodotto e quoziente.  
(data - ones, data * data, data / data)
```

```
⇒ (array([0, 1]), array([1, 4]), array([1., 1.]))
```


La libreria Numpy

Lo stesso avviene per differenza, prodotto e quoziente.



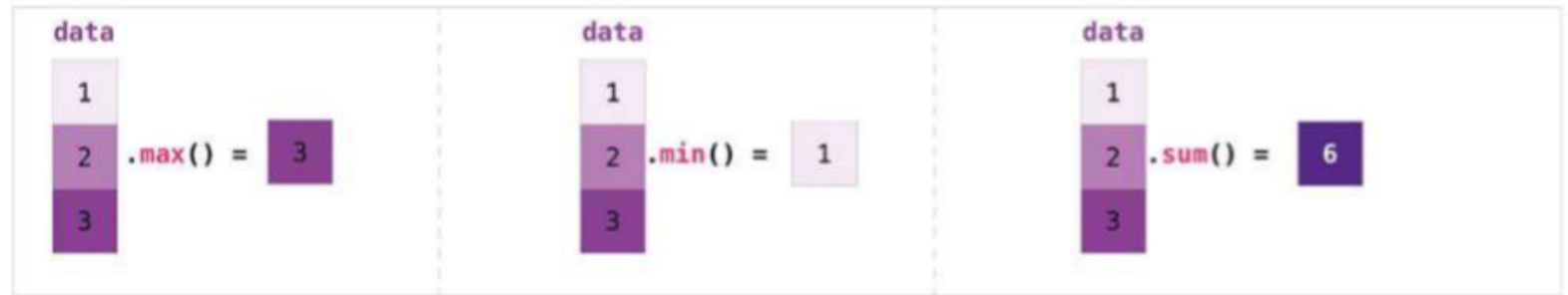
Questo comportamento delle operazioni viene detto **vettoriale**, quindi la somma vettoriale di due vettori significa che l'operazione viene automaticamente eseguita componente per componente, seguendo l'ordine con cui è stato creato l'array.

Oltre alle operazioni elementari, possiamo effettuare operazioni che «riassumono» il contenuto di un array, come la somma delle componenti di un vettore o la ricerca del massimo e del minimo di un array.

```
[12] data = np.array([1, 2, 3])  
  
[13] # Massimo, minimo e somma.  
      (data.max(), data.min(), data.sum())  
  
      (3, 1, 6)
```

La libreria Numpy

Per queste operazioni esistono metodi specifici che possiamo invocare sull'istanza. Infatti, non dobbiamo dimenticarci che **un array è un oggetto Numpy** e quindi, come un qualsiasi altro oggetto in Python, possiede **attributi** e **metodi**:

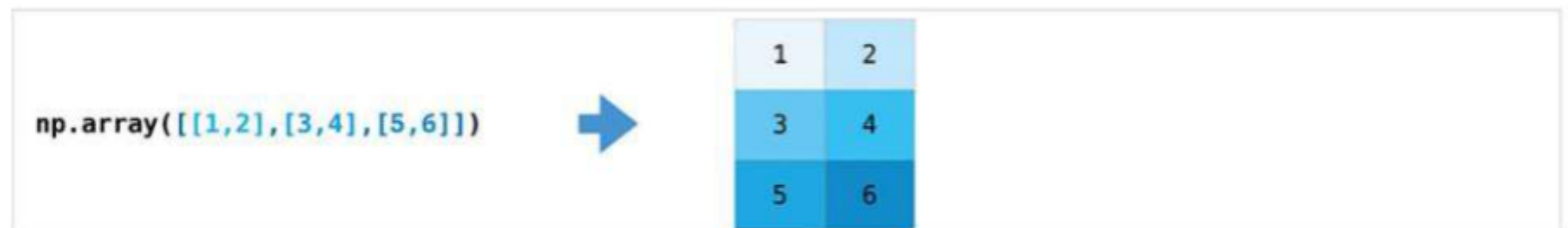


Se un vettore viene creato usando una lista, possiamo creare una matrice usando una lista di liste, come segue.

```
[14] data = np.array([
      [1, 2],
      [3, 4],
      [5, 6]
])

[15] data
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In questa rappresentazione, la lista più esterna rappresenta **una lista di righe**, mentre le liste più interne rappresentano i valori delle colonne.



La libreria Numpy

Indexing e slicing di una matrice funzionano in modo analogo a quelli di un vettore; l'unica differenza è data dal fatto che, visto che un array è bidimensionale, sono presenti due indici invece di uno solo: il primo indice seleziona le righe, mentre il secondo seleziona le colonne.

```
✓ [16] data[0, 1]
    ↗ 2

✓ [17] data[1:3]
    ↗ array([[3, 4],
             [5, 6]])

✓ [18] data[0:2, 0]
    ↗ array([1, 3])
```

Se il secondo indice non viene esplicitato, vengono selezionate automaticamente le righe, come segue.

data			data[0,1]			data[1:3]			data[0:2,0]		
	0	1		0	1		0	1		0	1
0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	1	3	4	1	3	4	1	3	4
2	5	6	2	5	6	2	5	6	2	5	6

La libreria Numpy

Allo stesso modo, le operazioni vettoriali vengono eseguite allineando le righe e le colonne ed eseguendo le operazioni elemento per elemento.

```
✓ [19] data, ones = np.array([[1, 2], [3, 4]]), np.array([[1, 1], [1, 1]])
```

```
✓ [20] (data, ones)
```

```
⇒ (array([[1, 2],  
         [3, 4]]),  
    array([[1, 1],  
         [1, 1]]))
```

```
✓ [21] data + ones
```

```
⇒ array([[2, 3],  
        [4, 5]])
```

$$\begin{array}{c} \text{data} \\ \text{data} + \text{ones} \end{array} = \begin{array}{|c|c|} \hline \text{data} & \\ \hline 1 & 2 \\ 3 & 4 \\ \hline \end{array} + \begin{array}{|c|c|} \hline \text{ones} & \\ \hline 1 & 1 \\ 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 3 \\ 4 & 5 \\ \hline \end{array}$$

Le operazioni di ricerca del massimo e del minimo si applicano anche nel caso delle matrici.

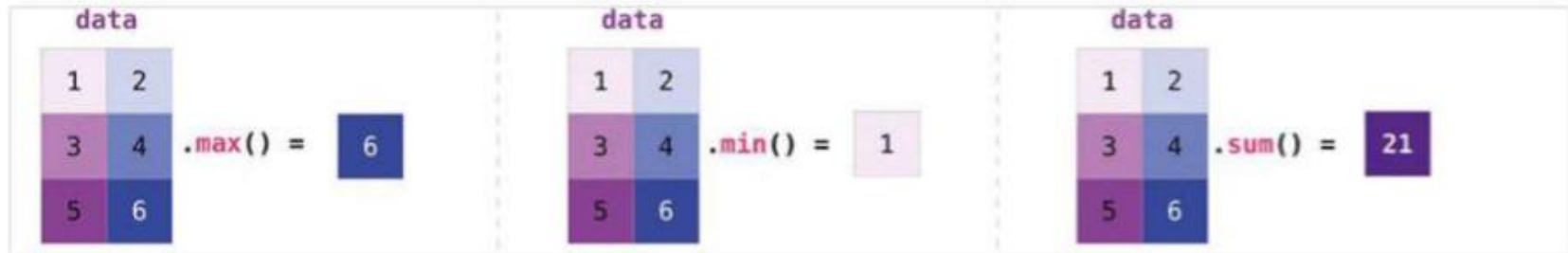
```
✓ [22] data = np.array([  
    [1, 2],  
    [3, 4],  
    [5, 6]  
])
```

```
✓ [23] (data.max(), data.min(), data.sum())
```

```
⇒ (6, 1, 21)
```

La libreria Numpy

In questo caso, è necessario fare particolare attenzione al verso, o **asse**, lungo cui applichiamo l'operazione. Se non specifichiamo l'asse, allora l'operazione verrà eseguita su tutti gli elementi che compongono la matrice.



Al contrario, se specifichiamo l'asse lungo cui calcolare il massimo, allora è possibile ottenere risultati per righe o per colonne configurando il valore del parametro opzionale `axis`. Se `axis` è zero allora si fa riferimento alle righe, mentre se `axis` è 1 allora il riferimento è alle colonne.

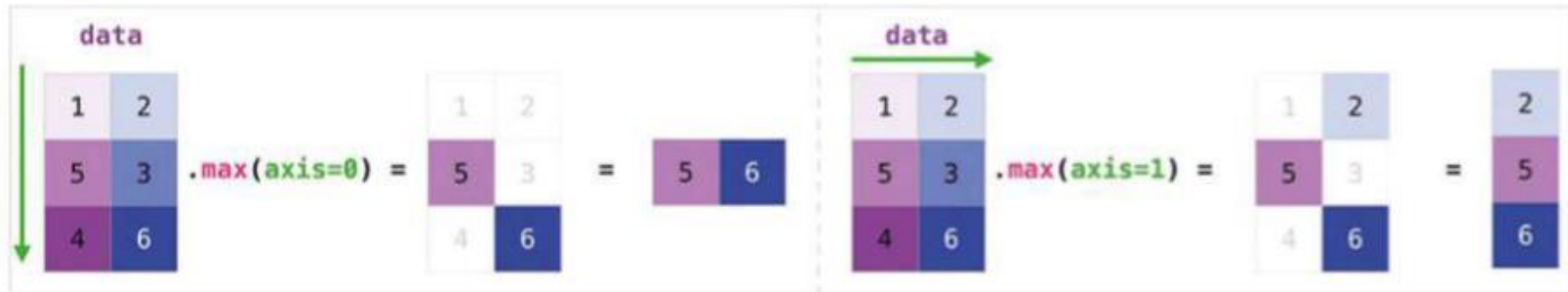
```
[24] data = np.array([
      [1, 2],
      [5, 3],
      [4, 6]
])

[25] (data.max(axis = 0), data.min(axis = 0), data.sum(axis = 0))
      (array([5, 6]), array([1, 2]), array([10, 11]))

[26] (data.max(axis = 1), data.min(axis = 1), data.sum(axis = 1))
      (array([2, 5, 6]), array([1, 3, 4]), array([ 3,  8, 10]))
```

La libreria Numpy

Per capire meglio la differenza tra righe e colonne, possiamo visualizzare l'effetto del parametro `axis` sul metodo `max`, come segue.



Un asse è una delle dimensioni di un array. Mentre nel caso di un vettore è presente un solo asse, per le matrici ce ne sono due, righe e colonne. Per array a più di due dimensioni di solito non si utilizzano nomi, ma numeri: gli assi vengono numerati a partire da 0 fino a $N-1$ dove N è il numero di dimensioni dell'array. Quindi, un vettore possiede solo l'asse 0, mentre le righe di una matrice corrispondono all'asse 0 e le colonne all'asse 1, e così via.

È possibile scambiare gli assi di una matrice (righe e colonne) grazie all'operatore `.T`, ottenendo così la **matrice trasposta**.

```
[27] data = np.array([
      [1, 2],
      [3, 4],
      [5, 6]
])

[28] (data, data.T)

(array([[1, 2],
       [3, 4],
       [5, 6]]),
 array([[1, 3, 5],
       [2, 4, 6]]))
```

La libreria Numpy

Di fatto, i valori della matrice non cambiano; quello che cambia è il modo in cui è possibile accedere a essi.

data

1	2
3	4
5	6

data.T

1	3	5
2	4	6

Se invece che scambiare gli assi vogliamo modificarne le lunghezze, possiamo utilizzare il metodo `reshape` per cambiare la forma dell'array, come segue.

```
[29] data = np.array([1, 2, 3, 4, 5, 6])  
  
[30] (data, data.reshape(2, 3), data.reshape(3, 2))  
  
(array([1, 2, 3, 4, 5, 6]),  
 array([[1, 2, 3],  
        [4, 5, 6]]),  
 array([[1, 2],  
        [3, 4],  
        [5, 6]]))
```

Ciò ci permette anche di passare **da vettore a matrice e viceversa**, a patto che il numero degli elementi della nuova forma sia lo stesso.

The diagram illustrates the process of reshaping a 1D array into 2D arrays. On the left, a vertical column labeled 'data' contains six elements: 1, 2, 3, 4, 5, and 6. The elements are color-coded in a gradient from light blue at the top to dark blue at the bottom. In the center, a 2x3 grid labeled 'data.reshape(2,3)' shows the first two rows of the original array. A red vertical bracket to its left indicates a height of 2, and a purple horizontal bracket below it indicates a width of 3. On the right, a 3x2 grid labeled 'data.reshape(3,2)' shows the first three rows of the original array. A red vertical bracket to its left indicates a height of 3, and a purple horizontal bracket below it indicates a width of 2.

data	data.reshape(2,3)	data.reshape(3,2)
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6

La libreria Numpy

Finora non abbiamo affrontato la questione della **forma** di un array.

Affinché si possa operare tra **array che hanno le stesse dimensioni**, per esempio vettore-vettore o matrice-matrice, è necessario che entrambi gli oggetti abbiano **la stessa forma**. Come abbiamo visto in queste pagine, le operazioni vettoriali vengono effettuate elemento per elemento, quindi dobbiamo poter operare sullo **stesso numero di elementi nella stessa posizione**, il che significa in pratica avere la stessa forma.

Esiste però un'eccezione a questa regola, che prende il nome di **ripetizione o broadcasting**.

In poche parole, è possibile eseguire delle operazioni tra due array se e solo se:

- i due array hanno la stessa forma, oppure
- una delle dimensioni è 1, **ripetendo**, cioè copiando, i valori lungo quella dimensione.

Vediamo il broadcasting in azione nel caso più semplice, la moltiplicazione vettore-scalare.



La libreria Numpy

```
✓ [31] data = np.array([1, 2])
0s

✓ [32] data * 1.6
0s
↔ array([1.6, 3.2])

✓ [33] data = np.array([
0s      [1, 2],
      [3, 4],
      [5, 6]
])

✓ [34] ones_row = np.array([1, 1])
0s

✓ ▶ data + ones_row
0s
↔ array([[2, 3],
        [4, 5],
        [6, 7]])
```

In questo caso, un singolo numero, detto **scalare**, viene *ripetuto* lungo l'asse 0 del vettore in modo da poter ottenere un nuovo vettore con la stessa forma del vettore per cui vogliamo moltiplicare lo scalare.

1	* 1.6	=	1	*	1.6	=	1.6
2			2		1.6		3.2

La libreria Numpy

Questa regola vale per tutti gli array, estendendo le normali operazioni matematiche elementari anche a casi più complessi, come nel caso della somma vettore-matrice, come segue.

$$\text{data} + \text{ones_row} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}$$

Qui il vettore riga viene ripetuto lungo le altre righe della matrice, ottenendo una nuova matrice con la stessa forma di quella originale.

La libreria Faker e la libreria Pandas

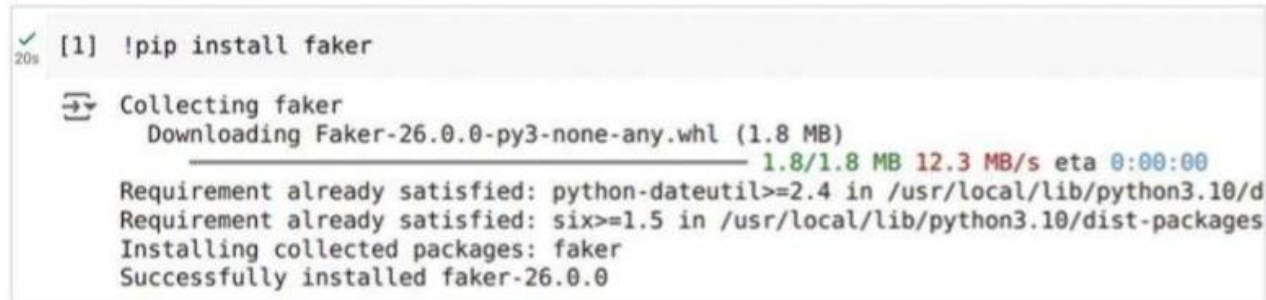
3 L'analisi di dati tabulari con *Pandas*

I dati tabulari rappresentano informazioni in un formato simile a quello delle matrici, che abbiamo appena visto. La differenza sta nel fatto che i valori della tabella non sono necessariamente tutti dello stesso tipo. Tipicamente, infatti, **sulle colonne si rappresentano le variabili della popolazione**, per esempio nome, età e indirizzo di un cliente, mentre **sulle righe ci sono le unità del campione** estratto dalla popolazione, ossia le singole osservazioni.

Per lavorare con i dati tabulari introduciamo una nuova libreria, chiamata **Faker** (<https://faker.readthedocs.io>), che utilizza un generatore di numeri casuali per simulare dei dati realistici come stringhe.

A differenza di *Numpy*, *Faker* non è una libreria presente di default nell'ambiente Colab, quindi è necessario installarla utilizzando il comando *pip*.

```
[1] !pip install faker
```



```
Collecting faker
  Downloading Faker-26.0.0-py3-none-any.whl (1.8 MB)
    1.8/1.8 MB 12.3 MB/s eta 0:00:00
Requirement already satisfied: python-dateutil>=2.4 in /usr/local/lib/python3.10/d
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages
Installing collected packages: faker
Successfully installed faker-26.0.0
```

Il punto esclamativo all'inizio della cella di codice permette di eseguire comandi direttamente nel terminale Colab invece che nell'interprete Python. A questo punto possiamo importare le librerie *Numpy*, *Pandas* e *Faker*. *Pandas* è la libreria che useremo per manipolare i dati che genereremo con *Faker*.

```
[2] import numpy as np
import pandas as pd
from faker import Faker
```

La libreria Faker e la libreria Pandas

```
✓ [2] import numpy as np  
0s      import pandas as pd  
      from faker import Faker
```

Come abbiamo visto nel precedente Paragrafo, il primo passo nell'utilizzo di un generatore di numeri pseudo-casuali consiste nell'inizializzazione del *seed*. Inoltre, *Faker* supporta la generazione di dati in diverse lingue (anche più lingue contemporaneamente). È possibile configurare la lingua di generazione quando invochiamo il costruttore del generatore, come nel codice che segue.

```
✓ [3] seed = 42                # Inizializzazione del seed.  
0s      Faker.seed(seed)        # Assegnazione del seed.  
      rng = Faker("it_IT")     # Costruzione del generatore.
```

Una volta costruito il generatore, per generare un valore casuale è sufficiente invocare uno dei tanti metodi disponibili per questo oggetto. Come nei generatori *Numpy* abbiamo invocato i metodi *random* e *integers* per generare un numero a virgola mobile o un numero intero, anche in *Faker* esistono metodi specifici a seconda del tipo di valore che vogliamo generare. Per esempio, il metodo *first_name* genera un nome di persona, mentre il metodo *city* genera un nome di città.

Per semplicità possiamo definire una funzione che accetta il generatore come parametro e restituisce un dizionario che rappresenta un cliente casuale.



Treecha/Shutterstock

La libreria Faker e la libreria Pandas

```
✓ [4] def rng_cliente(rng):  
0s     return {  
        "nome": rng.first_name(),  
        "cognome": rng.last_name(),  
        "città": rng.city(),  
        "cap": rng.postcode(),  
        "via": rng.street_address(),  
        "telefono": rng.phone_number()  
    }
```

Vediamo un esempio.

```
✓ [5] rng_cliente(rng)  
0s  
⇒ { 'nome': 'Eugenia',  
    'cognome': 'Caracciolo',  
    'città': 'Bianco',  
    'cap': '46100',  
    'via': 'Via Loredana, 18',  
    'telefono': '+39 0322338903' }
```

A questo punto possiamo generare una lista di clienti e visualizzare i primi tre.

```
✓ [6] clienti = []  
2s     for _ in range(0, 5000):  
        clienti.append(rng_cliente(rng))  
  
    clienti[:3]  
  
⇒ [{ 'nome': 'Ricciotti',  
    'cognome': 'Lettiere',  
    'città': 'Seriato',  
    'cap': '00043',  
    'via': 'Contrada Cremonesi, 5 Piano 1',  
    'telefono': '3534594071'},  
  { 'nome': 'Amanda',  
    'cognome': 'Magnani',  
    'città': 'Termini',  
    'cap': '19121',  
    'via': 'Canale Gianluigi, 1',  
    'telefono': '0536752551'},  
  { 'nome': 'Concetta',  
    'cognome': 'Donini',  
    'città': 'Corciano',  
    'cap': '00077',  
    'via': 'Contrada Volta, 5 Appartamento 55',  
    'telefono': '+39 04329537622'} ]
```


La libreria Faker e la libreria Pandas

Una volta che abbiamo generato i dati, possiamo procedere usando la libreria *Pandas* (<https://pandas.pydata.org>). Questa libreria si basa su un oggetto chiamato **DataFrame**, che rappresenta di fatto una tabella di dati. Per costruire l'oggetto basta chiamare il suo costruttore e passargli la lista di dizionari. In questo oggetto, **le chiavi dei dizionari corrispondono alle colonne della tabella**, mentre le righe sono dati dai valori dei dizionari.



```
✓ 0s ▶ clienti = pd.DataFrame(clienti)
```

clienti

	nome	cognome	città	cap	via	telefono
0	Ricciotti	Lettiere	Seriate	00043	Contrada Cremonesi, 5 Piano 1	3534594071
1	Amanda	Magnani	Termini	19121	Canale Gianluigi, 1	0536752551
2	Concetta	Donini	Corciano	00077	Contrada Volta, 5 Appartamento 55	+39 04329537622
3	Gioffre	Zamorani	Viazzano	93017	Borgo Cicilia, 1 Appartamento 22	0709784805
4	Costanzo	Magrassi	Osteria Del Gatto	00031	Contrada Temistocle, 98 Appartamento 32	+39 0965570153
...
4995	Elisa	Cignaroli	Castelfidardo	28823	Vicolo Eliana, 840	371349677
4996	Massimo	Pulci	Rovito	02016	Stretto Ennio, 8 Appartamento 5	034394742
4997	Iolanda	Barsanti	Sovere	20831	Piazza Calogero, 77	+39 04314315795
4998	Loretta	Boito	San Felice A Cancelli	26029	Incrocio Tatiana, 86 Appartamento 3	+39 0304376326
4999	Luisa	Garibaldi	San Giuseppe Jato	33036	Canale Paolo, 17 Appartamento 5	+39 043800767

5000 rows × 6 columns

La libreria Faker e la libreria Pandas

Ogni riga viene numerata in modo incrementale partendo da zero fino al numero massimo di elementi meno uno. Questa numerazione prende il nome di **indice**, che verrà utilizzato nella fase di accesso e manipolazione dei dati.

3.1 Input e output di dati tabulari

È possibile salvare su un file in formato CSV i dati che abbiamo appena generato con il metodo `to_csv`. Di solito, l'indice viene escluso dal salvataggio su file.

```
✓ [8] clienti.to_csv("clienti.csv", index = False)
```

Se apriamo il file con l'I/O di Python vediamo che la prima riga contiene l'intestazione delle colonne e i dati partono dalla seconda riga del file.

```
✓ [9] f = open("clienti.csv", "r")
      print(f.readline())
      print(f.readline())
      f.close()

nome,cognome,città,cap,via,telefono
Ricciotti,Lettiere,Seriate,00043,"Contrada Cremonesi, 5 Piano 1",3534594071
```

Per leggere il file CSV possiamo usare la funzione `pd.read_csv`. Per confrontarlo con il DataFrame in memoria possiamo impostare il tipo di lettura a stringhe e utilizzare il metodo `equals`.

```
✓ [10] assert(clienti.equals(pd.read_csv("clienti.csv", dtype = "str")))
```

In questo modo non dobbiamo scorrere manualmente i file, ma possiamo comodamente importare dati dall'esterno utilizzando le funzioni di *Pandas*.

La libreria Faker e la libreria Pandas

3.2 L'accesso localizzato

Possiamo visualizzare e accedere ai dati contenuti in un DataFrame in diversi modi. Per un controllo veloce della struttura dei dati, possiamo utilizzare i metodi **head** e **tail**, che visualizzano rispettivamente le prime cinque righe e le ultime cinque righe del DataFrame.

```
✓ [11] clienti.head()
```



	nome	cognome	città	cap	via	telefono
0	Ricciotti	Lettiere	Seriate	00043	Contrada Cremonesi, 5 Piano 1	3534594071
1	Amanda	Magnani	Termini	19121	Canale Gianluigi, 1	0536752551
2	Concetta	Donini	Corciano	00077	Contrada Volta, 5 Appartamento 55	+39 04329537622
3	Gioffre	Zamorani	Viazzano	93017	Borgo Cicilia, 1 Appartamento 22	0709784805
4	Costanzo	Magrassi	Osteria Del Gatto	00031	Contrada Temistocle, 98 Appartamento 32	+39 0965570153

```
✓ [12] clienti.tail()
```



	nome	cognome	città	cap	via	telefono
4995	Elisa	Cignaroli	Castelfidardo	28823	Vicolo Eliana, 840	371349677
4996	Massimo	Pulci	Rovito	02016	Stretto Ennio, 8 Appartamento 5	034394742
4997	Iolanda	Barsanti	Sovere	20831	Piazza Calogero, 77	+39 04314315795
4998	Loretta	Boito	San Felice A Cancellò	26029	Incrocio Tatiana, 86 Appartamento 3	+39 0304376326
4999	Luisa	Garibaldi	San Giuseppe Jato	33036	Canale Paolo, 17 Appartamento 5	+39 043800767

Se vogliamo accedere arbitrariamente a righe e colonne dobbiamo basarci sui valori dell'indice e sui nomi delle colonne, come segue.

La libreria Faker e la libreria Pandas

```
✓ [13] clienti.index
```

```
⇒ RangeIndex(start=0, stop=5000, step=1)
```

```
✓ [14] clienti.columns
```

```
⇒ Index(['nome', 'cognome', 'città', 'cap', 'via', 'telefono'], dtype='object')
```

Per esempio, se vogliamo accedere alla colonna nome è sufficiente passare questa stringa al DataFrame tra parentesi quadre.

```
✓ [15] clienti["nome"]
```

```
⇒ 0      Ricciotti  
   1      Amanda  
   2      Concetta  
   3      Gioffre  
   4      Costanzo  
   ...  
4995      Elisa  
4996      Massimo  
4997      Iolanda  
4998      Loretta  
4999      Luisa  
Name: nome, Length: 5000, dtype: object
```

La libreria Faker e la libreria Pandas

Mentre se volessimo un sottoinsieme delle righe ci basterebbe passare un intervallo di valori, come abbiamo visto per gli array.

```
✓ [16] clienti[5:12]
```



	nome	cognome	città	cap	via	telefono
5	Gioacchino	Zola	Ruvo Del Monte	84075	Stretto Marisa, 92 Piano 8	37763834671
6	Gioffre	Gasperi	Caporosso	13853	Vicolo Giancarlo, 1	+39 3511051837
7	Adriana	Cherubini	Paina	26833	Borgo Caccioppoli, 786 Appartamento 1	34351333822
8	Gustavo	Scandone	Altofonte	80100	Piazza Fernanda, 1 Piano 7	04326026044
9	Piersanti	Vivaldi	Tremignon	02043	Borgo Biagi, 5	086282081
10	Carolina	Boldù	Pomaretto	74018	Vicolo Carolina, 99	37114353427
11	Luchino	Bonolis	Obermais	97019	Stretto Giampaolo, 22	0541278491

Per combinare i due approcci dobbiamo chiamare il metodo `loc` e passare in ordine prima l'intervallo per le righe e poi il nome delle colonne.

```
✓ [17] clienti.loc[5:12, "nome"]
```



```
5    Gioacchino
6      Gioffre
7      Adriana
8      Gustavo
9    Piersanti
10   Carolina
11    Luchino
12      Tina
Name: nome, dtype: object
```

La libreria Faker e la libreria Pandas

Il metodo `loc` è molto flessibile e accetta anche una lista di colonne per estrarre un sottoinsieme del DataFrame.

```
[18] clienti.loc[5:12, ["nome", "cognome", "telefono"]]
```

	nome	cognome	telefono
5	Gioacchino	Zola	37763834671
6	Gioffre	Gasperi	+39 3511051837
7	Adriana	Cherubini	34351333822
8	Gustavo	Scandone	04326026044
9	Piersanti	Vivaldi	086282081
10	Carolina	Boldù	37114353427
11	Luchino	Bonolis	0541278491
12	Tina	Chindamo	05001640022

3.3 La selezione dei dati

Oltre a poter accedere ai dati tramite indice e colonne, è possibile definire delle condizioni e selezionare solo i valori che le soddisfano. Questo metodo è basato sulla definizione di **maschere binarie** di selezione, ossia sequenze di valori vero o falso che vengono utilizzate per selezionare le righe.

Se, per esempio, volessimo selezionare tutti i clienti che si chiamano "Maria" dovremmo costruire una maschera binaria con l'operatore di confronto `==` applicato alla colonna `nome`.

```
[19] clienti["nome"] == "Maria"
```

0	False
1	False
2	False
3	False
4	False
...	...
4995	False
4996	False
4997	False
4998	False
4999	False

Name: nome, Length: 5000, dtype: bool

La libreria Faker e la libreria Pandas

Una volta definita la maschera binaria, questa può essere utilizzata per accedere alle righe. In questo modo, verranno selezionate solo le righe che soddisfano questa condizione, e cioè che presentano il valore True nella maschera.

```
✓ [20] clienti[clienti["nome"] == "Maria"].head()
```

	nome	cognome	città	cap	via	telefono
181	Maria	Squarcione	Castellanza	81014	Borgo Bataglia, 788 Appartamento 20	37774989269
192	Maria	Morosini	Balbiano	91017	Strada Guglielmi, 6	0736588041
285	Maria	Mazzocchi	Capodarco	52047	Viale Mariana, 4	08516479434
353	Maria	Contarini	Alessandria Della Rocca	26024	Viale Gianpaolo, 247	3246377802
548	Maria	Foscari	San Rocchetto	56028	Incrocio Federico, 891 Appartamento 7	+39 0706615544

Quando è necessario confrontare più valori è possibile utilizzare il metodo `isin`, che restituisce vero se il valore della riga è contenuto all'interno di una **lista di selezione**.

```
✓ [21] clienti[clienti["nome"].isin(["Maria", "Luca"])].head()
```

	nome	cognome	città	cap	via	telefono
181	Maria	Squarcione	Castellanza	81014	Borgo Bataglia, 788 Appartamento 20	37774989269
192	Maria	Morosini	Balbiano	91017	Strada Guglielmi, 6	0736588041
285	Maria	Mazzocchi	Capodarco	52047	Viale Mariana, 4	08516479434
313	Luca	Santorio	Colpalombo	20094	Borgo Fiorino, 299 Appartamento 20	+39 03977697461
353	Maria	Contarini	Alessandria Della Rocca	26024	Viale Gianpaolo, 247	3246377802


Le maschere binarie rappresentano un approccio molto flessibile alla selezione dei dati e permettono di sfruttare le stesse condizioni che utilizziamo normalmente in Python con la struttura `if-else`.

La libreria Faker e la libreria Pandas

3.4 La manipolazione dei dati

Oltre a estrarre dei dati, è possibile aggiungerne di nuovi e manipolare quelli esistenti. Per esempio, è possibile aggiungere una nuova colonna nello stesso modo in cui aggiungiamo un valore a un dizionario.

```
✓ [22] data_di_nascita = []  
0s   for _ in range(len(clienti)):  
       data_di_nascita.append(  
           rng.date_of_birth(  
               minimum_age = 18,  
               maximum_age = 95  
           )  
       )  
       clienti["data_di_nascita"] = data_di_nascita  
  
       clienti.head()
```



	nome	cognome	città	cap	via	telefono	data_di_nascita
0	Ricciotti	Lettiere	Seriate	00043	Contrada Cremonesi, 5 Piano 1	3534594071	1990-06-10
1	Amanda	Magnani	Termini	19121	Canale Gianluigi, 1	0536752551	1929-10-23
2	Concetta	Donini	Corciano	00077	Contrada Volta, 5 Appartamento 55	+39 04329537622	1964-09-29
3	Gioffre	Zamorani	Viazzano	93017	Borgo Cicilia, 1 Appartamento 22	0709784805	1939-07-27
4	Costanzo	Magrassi	Osteria Del Gatto	00031	Contrada Temistocle, 98 Appartamento 32	+39 0965570153	1942-01-23

Python supporta nativamente il tipo di valore data, che è composto da una tupla di tre numeri (rispettivamente anno, mese e giorno).

```
✓ [23] clienti.loc[0, "data_di_nascita"]  
0s  
  
🔍 datetime.date(1990, 6, 10)
```


La libreria Faker e la libreria Pandas

Pandas fornisce il metodo `apply` che accetta in input una funzione e restituisce una nuova sequenza di dati generata dall'applicazione della funzione a una colonna.

```
✓ [24] eta = clienti["data_di_nascita"].apply(lambda x: 2025 - x.year)
```

eta

```
0 35
1 96
2 61
3 86
4 83
..
4995 56
4996 62
4997 22
4998 65
4999 86
```

Name: data_di_nascita, Length: 5000, dtype: int64

Visto che questi dati sono vettoriali, è possibile applicare le funzioni vettoriali, come massimo, minimo e media.

```
✓ [25] (eta.min(), eta.max(), eta.mean())
```

```
(19, 97, 57.891)
```

Nello stesso modo è possibile salvare nel `DataFrame` i dati manipolati per poterli poi usare successivamente.

```
✓ [26] clienti["eta"] = eta
```

La libreria Faker e la libreria Pandas

3.5 I dati mancanti

Quando si manipolano dei dati, un tema fondamentale è quello dei dati mancanti. Possiamo simulare la perdita di dati sempre facendo affidamento a un generatore di numeri pseudo-casuali.

Iniziamo innanzi tutto generando una maschera binaria.



greenbutterfly/Shutterstock

```
✓ [27] mancanti = np.random.default_rng(seed).choice([False, True], len(clienti))  
0s  
      mancanti[:5]  
  
→ array([False,  True,  True, False, False])
```

Rimuoviamo quindi i dati dalla colonna telefono, utilizzando il valore `pd.NA`, dove NA sta per *Not available*, ossia non disponibile, mancante.

```
✓ [28] clienti.loc[mancanti, "telefono"] = pd.NA  
0s  
      clienti["telefono"].head()  
  
→ 0      3534594071  
   1              <NA>  
   2              <NA>  
   3      0709784805  
   4 +39 0965570153  
   Name: telefono, dtype: object
```

La libreria Faker e la libreria Pandas

È possibile controllare la presenza di dati mancanti con la funzione `isna`.

✓ [29] clienti.isna()

0s

	nome	cognome	città	cap	via	telefono	data_di_nascita	età
0	False	False	False	False	False	False	False	False
1	False	False	False	False	False	True	False	False
2	False	False	False	False	False	True	False	False
3	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False
...
4995	False	False	False	False	False	False	False	False
4996	False	False	False	False	False	True	False	False
4997	False	False	False	False	False	True	False	False
4998	False	False	False	False	False	True	False	False
4999	False	False	False	False	False	False	False	False

5000 rows × 8 columns

Combinando quello che abbiamo visto finora, immaginiamo di voler effettuare una campagna telefonica che ha come obiettivo i clienti *junior*, ovvero coloro che hanno al massimo 25 anni. Per selezionare questo sottoinsieme di clienti, dobbiamo innanzi tutto imporre un vincolo sull'età; per evitare di considerare anche utenti che non abbiamo la possibilità di contattare, dovremo quindi scartare coloro che non hanno un numero di telefono.

La libreria Faker e la libreria Pandas

```
✓ [30] clienti_junior = clienti[clienti["eta"] <= 25]
0s clienti_junior[["nome", "cognome", "telefono"]].dropna()
```

	nome	cognome	telefono
8	Gustavo	Scandone	04326026044
35	Francesco	Travaglia	+39 056480861
95	Ida	Fittipaldi	0565609328
99	Carmelo	Pisacane	053201489
149	Silvia	Gabba	+39 04444944654
...
4928	Federica	Alboni	+39 3811532270
4949	Beatrice	Tomei	+39 05876559475
4965	Martino	Pepe	+39 09539300344
4971	Pomponio	Morlacchi	3792129778
4983	Lisa	Barsanti	0961597821

234 rows x 3 columns

Esistono diverse strategie per gestire i dati mancanti, in questo esempio abbiamo semplicemente scartato dal campione quei clienti che non hanno fornito un recapito telefonico, dato che non erano rilevanti per il fine dell'analisi, ma in altri casi potrebbe essere possibile recuperare i dati non disponibili.

Le operazioni che abbiamo visto in questo Paragrafo sono disponibili all'interno del notebook *analisi_dati.ipynb*.

La libreria Matplotlib

4 Visualizzare i dati con *Matplotlib*

4.1 La visualizzazione base

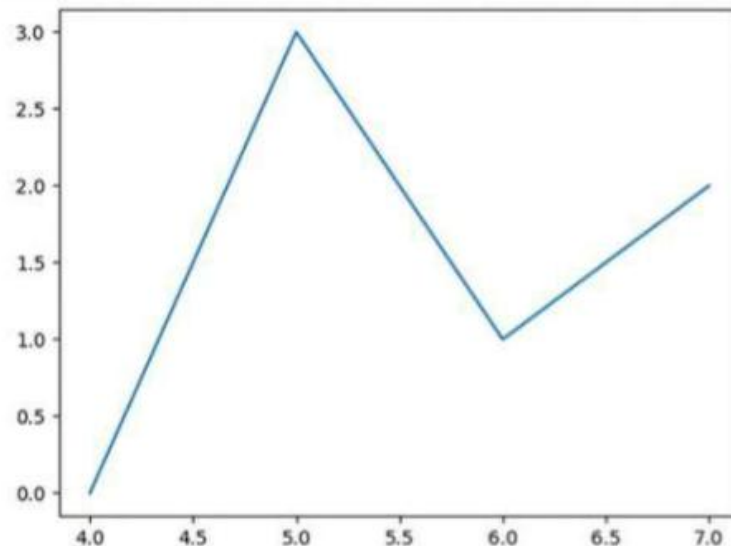
Per visualizzare dati in un progetto Python utilizzeremo la libreria *Matplotlib* (<https://matplotlib.org>), che è già inclusa in Colab.

```
[1] import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Grazie a questa libreria è possibile visualizzare dei dati su un piano cartesiano semplicemente fornendo le coppie di coordinate, come segue.

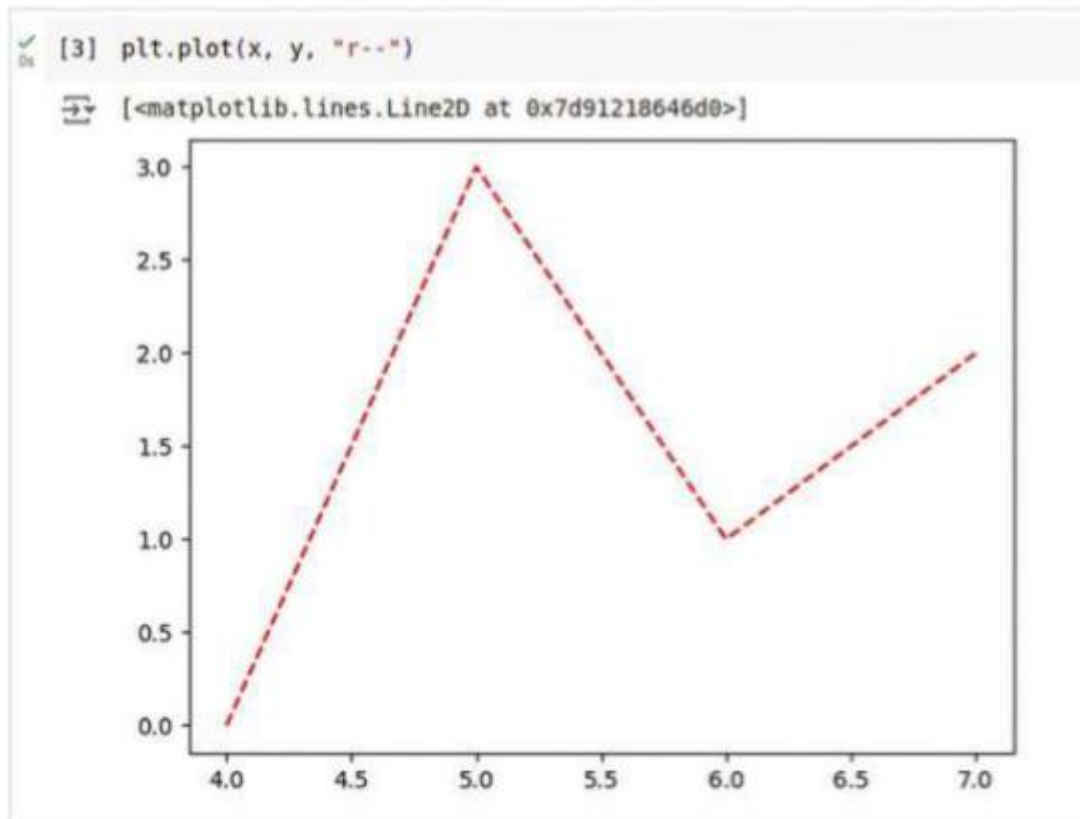
```
x = np.array([4, 5, 6, 7])
y = np.array([0, 3, 1, 2])
plt.plot(x, y)
```

[<matplotlib.lines.Line2D at 0x7d912392e650>]



La libreria Matplotlib

La funzione **plt.plot** collega automaticamente le coordinate che vengono inserite, formando una linea spezzata che mostra l'andamento dei valori. La funzione **plt.plot** supporta molti parametri opzionali (qui ne vedremo solo alcuni). Oltre alle coordinate, il primo parametro opzionale consente di **personalizzare lo stile** di visualizzazione utilizzando una stringa di combinazione di **colore-tratto**. Per esempio **r - -** sta per rosso (red) e tratteggiato.

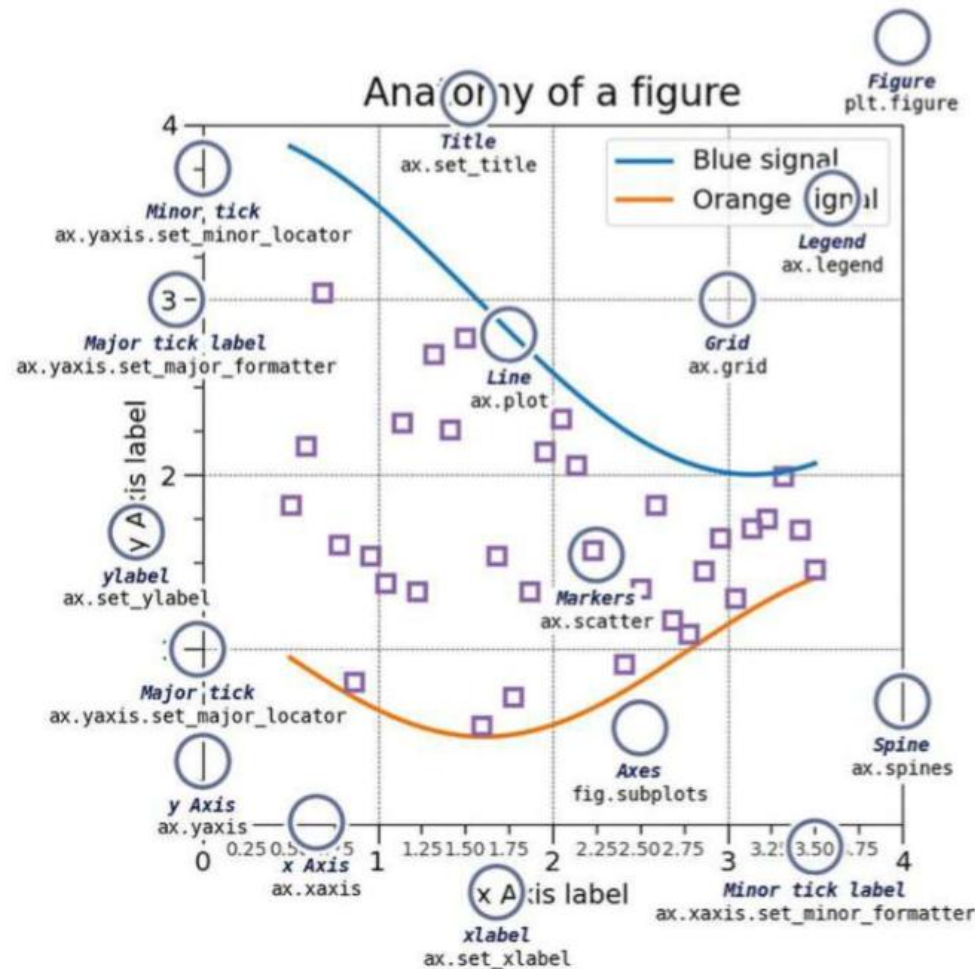


La libreria Matplotlib

4.2 Anatomia di una figura

La funzione `plt.plot` è utile per visualizzare dati velocemente senza dover utilizzare combinazioni complesse di oggetti e metodi. Per comprendere meglio il funzionamento di *Matplotlib* dobbiamo conoscere l'anatomia del suo oggetto **Figure**.

Fig. 13 mostra a colpo d'occhio come è strutturata una figura.



La libreria Matplotlib

In una figura ogni elemento assolve a una funzione specifica:

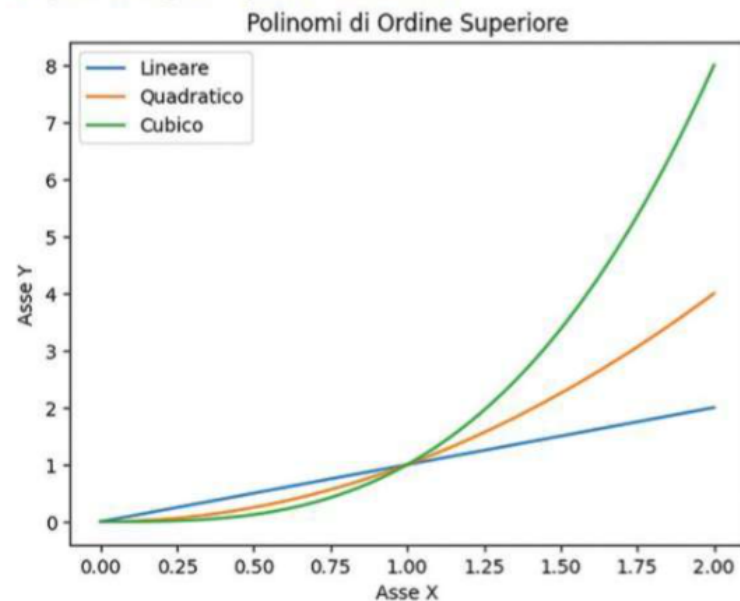
- `plt.figure` alloca memoria per la figura;
- `fig.subplots` inizializza il piano cartesiano;
- `ax.plot` permette di visualizzare dati sul piano cartesiano;
- `ax.scatter` visualizza i punti sul piano;
- `ax.set_title` imposta i titola della figura;
- `ax.set_xlabel` e `ax.set_ylabel` impostano le etichette degli assi;
- `ax.legend` imposta la legenda della figura.

Sono poi disponibili altri elementi, dettagliati nella documentazione ufficiale.

Vediamo ora come creare una figura più elaborata di quelle precedenti.

```
[6] # Creazione dati.  
x = np.linspace(0, 2, 100)  
# Creazione figura e assi.  
fig, ax = plt.subplots()  
# Aggiunta dati.  
ax.plot(x, x, label = "Lineare")  
ax.plot(x, x ** 2, label = "Quadratico")  
ax.plot(x, x ** 3, label = "Cubico")  
# Aggiunta descrizione.  
ax.set_title("Polinomi di Ordine Superiore")  
ax.set_xlabel("Asse X")  
ax.set_ylabel("Asse Y")  
ax.legend()
```

<matplotlib.legend.Legend at 0x7d91215d6500>



La libreria Matplotlib

Notiamo che la figura è creata utilizzando la funzione `plt.subplots`, che, come si può intuire dal nome, è in grado di generare più plot alla volta. In questo caso, non specificando nessun parametro opzionale, verrà creata una sola figura. La funzione restituisce una tupla di due valori, `fig` (ovvero *figure*) e `ax` (che sta per *axes*). Le stesse funzioni che avremmo prima eseguito da `plt`, ora le possiamo eseguire tramite l'oggetto `ax`. In particolare:

- `plot` si comporta esattamente allo stesso modo di `plt.plot`, con l'unica differenza che è possibile decidere a quale figura assegnare una visualizzazione (aspetto molto utile quando ci sono più figure da visualizzare in contemporanea);
- `set_title`, `set_xlabel` e `set_ylabel` servono a impostare rispettivamente il nome della figura e i nomi degli assi;
- `legend` aggiunge una legenda in base ai nomi e ai colori dei dati che sono stati aggiunti tramite `plot`.

4.3 La visualizzazione di dati tabulari

Matplotlib è integrato direttamente in *Pandas*, cosa che semplifica la visualizzazione dei dati tabulari, evitando di dover gestire le figure a basso livello. Per esempio, inizializziamo un *DataFrame* con l'andamento annuale delle temperature.

```
[7] data = pd.DataFrame({
    "anno": [2015, 2016, 2017, 2018, 2019, 2020],
    "t_max": [30, 35, 32, 40, 41, 42],
    "t_min": [-5, -2, -1, -7, -10, -6]
})

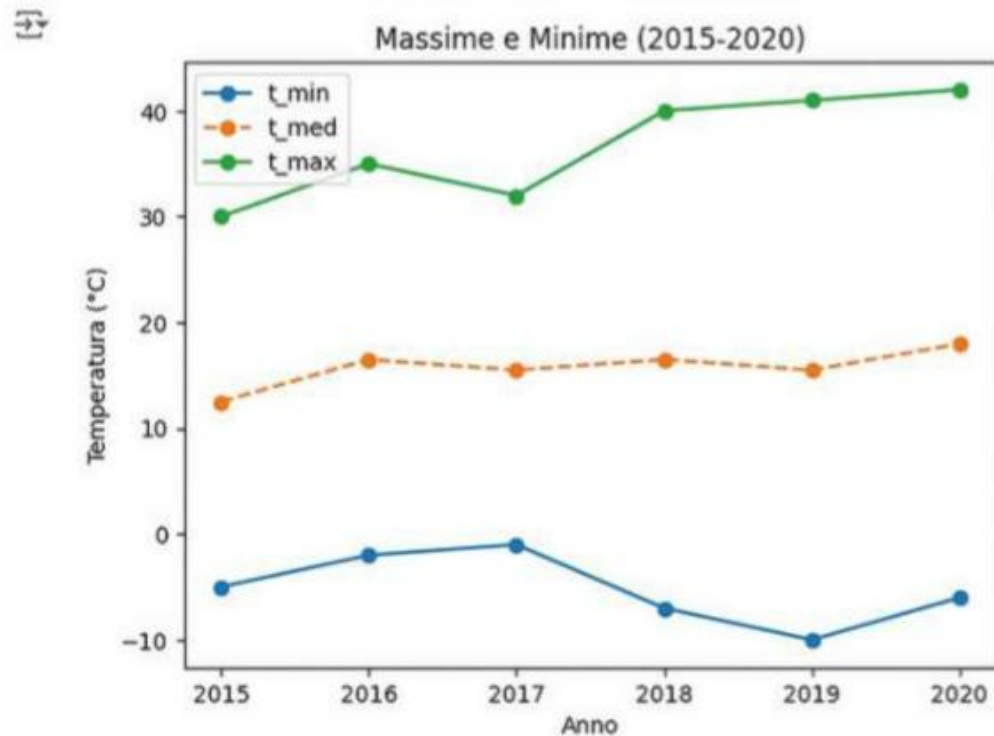
data
```

	anno	t_max	t_min
0	2015	30	-5
1	2016	35	-2
2	2017	32	-1
3	2018	40	-7
4	2019	41	-10
5	2020	42	-6

La libreria Matplotlib

Possiamo accedere all'interfaccia di visualizzazione direttamente dal metodo `plot` del `DataFrame`, sfruttando i parametri opzionali di *Matplotlib*.

```
[8] # Calcolo della temperatura media.  
data["t_med"] = (data["t_max"] + data["t_min"]) / 2  
# Visualizzazione andamento annuale.  
_ = data.plot(  
    x = "anno",  
    y = ["t_min", "t_med", "t_max"],  
    style = ["o-", "o--", "o-"],  
    title = "Massime e Minime (2015-2020)",  
    xlabel = "Anno",  
    ylabel = "Temperatura (°C)"  
)
```



Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

In questa Esercitazione guidata riprenderemo i concetti che abbiamo visto nel Capitolo e li applicheremo agli Open Data messi a disposizione dal Comune di Milano. In particolare, utilizzeremo i dati sulle colonnine di ricarica per veicoli elettrici presenti nel territorio comunale e li visualizzeremo come dati geospaziali. Per farlo faremo uso della libreria *GeoPandas*, che sarà necessaria insieme ad altre librerie di sistema utili per scaricare file dal web.

1 Importiamo le librerie necessarie.

```
✓ [1] import pandas as pd
1s      import matplotlib.pyplot as plt

      # Visualizzazione mappa.
      import io
      import requests
      import zipfile
      import geopandas as gpd
```

2 Scarichiamo i dati.

Il primo file che scarichiamo è un file CSV che contiene le informazioni sulle colonnine di ricarica per veicoli elettrici. Dato che l'URL del file è molto lungo, lo suddividiamo in diverse parti per visualizzare meglio i vari componenti, poi uniamo le varie stringhe con il metodo `join`.

```
✓ [2] url = "".join([
0s      "https://dati.comune.milano.it/",
      "dataset/0c7a321d-6055-4eed-bfb0-9bd2a8dabf88/",
      "resource/d9509257-9f88-4729-8905-c92f5bbda35f/"
      "download/ricarica_colonnine.csv"
    ])
```

Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

3 Prepariamo i dati per l'elaborazione.

Pandas è in grado di aprire i file CSV online in autonomia. Per farlo, è sufficiente passare l'URL come primo argomento della funzione `pd.read_csv`.

Visto che le informazioni contenute nel `DataFrame` sono tante, ci limiteremo a estrarre il nome del zona, la tipologia di colonnina, l'anno di installazione e la sua posizione, espressa attraverso le coordinate GPS.

```
[3] data = pd.read_csv(url, sep = ";")
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 239 entries, 0 to 238
Data columns (total 24 columns):
#   Column                Non-Null Count  Dtype
---  -
0   id_amat                239 non-null   int64
1   municipio              239 non-null   int64
2   id_sottonil            239 non-null   int64
3   id_nil                 239 non-null   int64
4   nome_nil               239 non-null   object
5   cerchia                239 non-null   object
6   ambito_id              216 non-null   float64
7   ambito_nome            216 non-null   object
8   attuazione            239 non-null   int64
9   tipologia              239 non-null   object
10  titolare               239 non-null   object
11  id_via                 239 non-null   int64
12  nome_via               239 non-null   object
13  localita               239 non-null   object
14  numero_col             239 non-null   int64
15  numero_pdr             239 non-null   int64
16  infra                  239 non-null   object
17  progetto               238 non-null   object
18  note                   239 non-null   object
19  ordinanza              239 non-null   object
20  anno                   238 non-null   float64
21  LONG_X_4326            239 non-null   float64
22  LAT_Y_4326             239 non-null   float64
23  Location                239 non-null   object
dtypes: float64(4), int64(8), object(12)
memory usage: 44.9+ KB
```

Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

Estraiamo ora i dati che ci interessano.

```
✓ [4] data = data[["ambito_nome", "tipologia", "anno", "Location"]]  
0s data.head()
```

	ambito_nome	tipologia	anno	Location
0	Centro storico	QN	2022.0	(45.4621490324913, 9.19357713616389)
1	Centro storico	Q	2021.0	(45.467629357113, 9.17760367245734)
2	Lazzaretto - Buenos Aires	QN	NaN	(45.4844835706687, 9.20490805319666)
3	Argonne	N	2022.0	(45.4842765276858, 9.23541763461848)
4	Abbiategrasso	N	2022.0	(45.437301971912, 9.16916994362816)

Notiamo che l'anno è stato convertito in un numero in virgola mobile e la colonna contiene dei dati mancanti. Nessuna delle due cose è un problema: *Pandas* supporta un formato di numero intero in grado di gestire correttamente i dati mancanti, mentre per convertire una colonna da tipo a un altro tipo possiamo utilizzare il metodo `astype` e passare la stringa che indica il tipo da utilizzare.

```
✓ data["anno"] = data["anno"].astype("Int64")  
0s data.head()
```

	ambito_nome	tipologia	anno	Location
0	Centro storico	QN	2022	(45.4621490324913, 9.19357713616389)
1	Centro storico	Q	2021	(45.467629357113, 9.17760367245734)
2	Lazzaretto - Buenos Aires	QN	<NA>	(45.4844835706687, 9.20490805319666)
3	Argonne	N	2022	(45.4842765276858, 9.23541763461848)
4	Abbiategrasso	N	2022	(45.437301971912, 9.16916994362816)

Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

- 4 Visualizziamo con un grafico a torta la distribuzione delle colonnine.
- Ora che abbiamo i dati nel formato corretto, possiamo invocare innanzi tutto il metodo `value_counts` sulla colonna dei quartieri per contare quante colonnine sono state installate. Per facilitare la visualizzazione, selezioniamo solo le voci che hanno almeno 8 colonnine.

```
✓ [6] totale_nome = data["ambito_nome"].value_counts()  
0s totale_nome_min_8 = totale_nome[totale_nome >= 8]
```

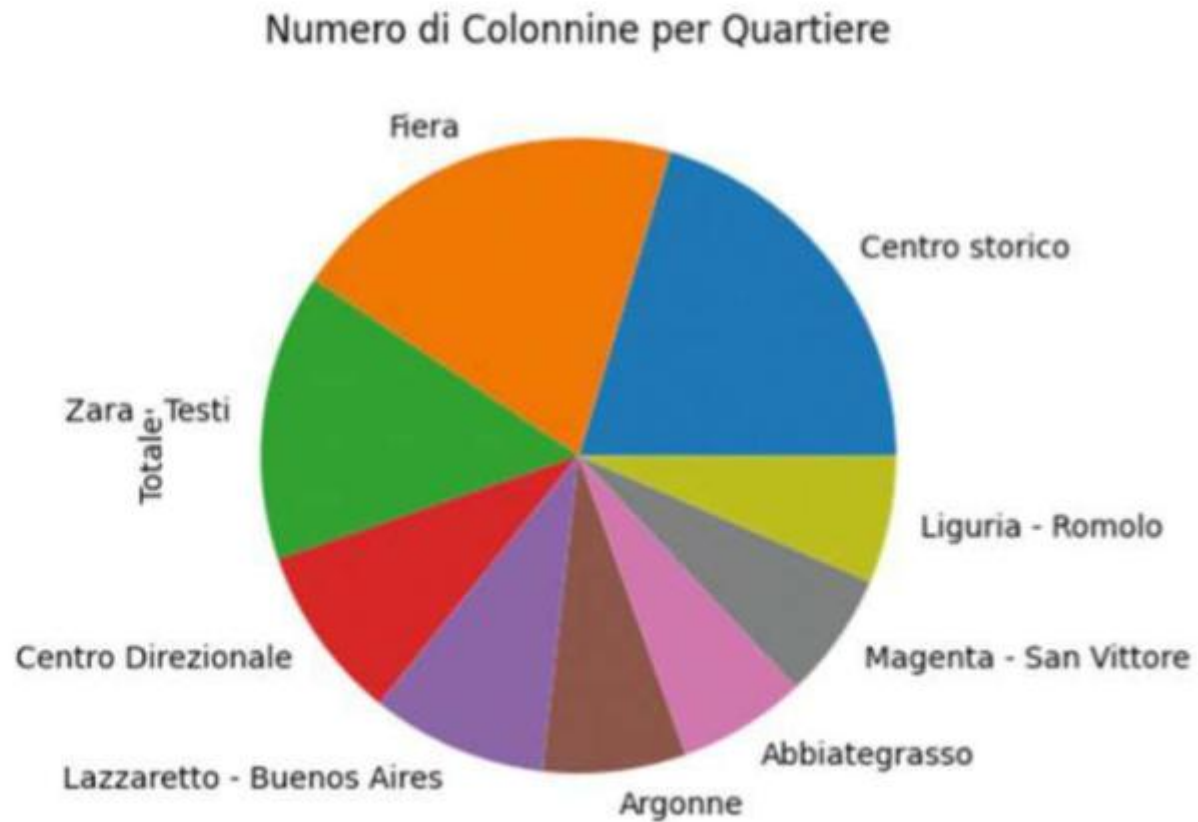
```
totale_nome_min_8
```

```
⇒ ambito_nome  
Centro storico      25  
Fiera              25  
Zara - Testi       18  
Centro Direzionale 11  
Lazzaretto - Buenos Aires 11  
Argonne            9  
Abbiategrasso      8  
Magenta - San Vittore 8  
Liguria - Romolo    8  
Name: count, dtype: int64
```

Esercitazione gi

A questo punto possiamo visualizzare la distribuzione delle colonnine con un grafico a torta.

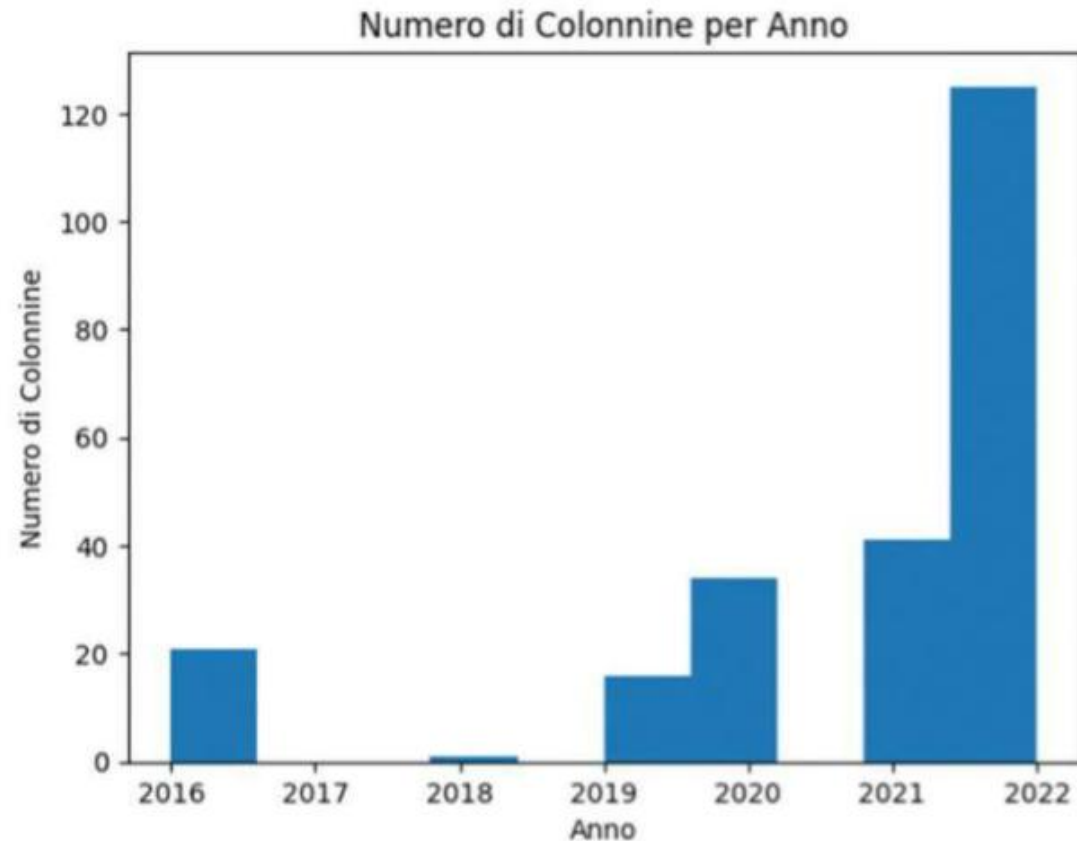
```
✓ [7] _ = totale_nome_min_8.plot(  
0s kind = "pie",  
title = "Numero di Colonnine per Quartiere",  
ylabel = "Totale"  
)
```



Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

5 Visualizziamo con un istogramma le colonnine installate per anno.

```
✓ [8] _ = data["anno"].plot(  
0s     kind = "hist",  
       title = "Numero di Colonnine per Anno",  
       xlabel = "Anno",  
       ylabel = "Numero di Colonnine"  
)
```



Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

⑥ Combiniamo le due visualizzazioni.

Se vogliamo combinare le due visualizzazioni, per prima cosa dobbiamo utilizzare il metodo `groupby`, che raggruppa le righe per zona e poi conta le colonnine per anno in quella determinata zona.

```
✓ [9] totale_gruppo = data[["ambito_nome", "anno"]].groupby(["ambito_nome"]).value_counts()
```

totale_gruppo

ambito_nome	anno	
Abbiategrasso	2022	6
	2020	1
	2021	1
Argonne	2022	7
	2016	1
Zara - Testi	2022	10
	2020	3
	2019	2
	2021	2
	2016	1

Name: count, Length: 89, dtype: int64

L'oggetto `groupby` può essere convertito a un normale `DataFrame` con il metodo `to_frame`, specificando il nome della colonna calcolata da `value_counts`. Dato che abbiamo raggruppato contemporaneamente sia sulla zona sia sull'anno, l'indice risultante è un indice composto, che potrebbe essere difficile da utilizzare. Per ovviare a questo problema è sufficiente «resettare» l'indice e trasformarlo in colonne con il metodo `reset_index`. Alla fine quello che otteniamo è un semplice `DataFrame` con i conteggi per zona e anno.

Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

```
✓ [10] totale_gruppo = totale_gruppo.to_frame("totale").reset_index()
0s
```

totale_gruppo

	ambito_nome	anno	totale
0	Abbiategrasso	2022	6
1	Abbiategrasso	2020	1
2	Abbiategrasso	2021	1
3	Argonne	2022	7
4	Argonne	2016	1
...
84	Zara - Testi	2022	10
85	Zara - Testi	2020	3
86	Zara - Testi	2019	2
87	Zara - Testi	2021	2
88	Zara - Testi	2016	1

89 rows × 3 columns

Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

Dato che di solito le funzioni di visualizzazione si aspettano in ingresso un indice e delle colonne, possiamo trasformare il DataFrame con il metodo pivot, utilizzando gli anni come indice, le zone come colonne e i conteggi del totale come valori delle righe.

```
[11] totale_gruppo = totale_gruppo.pivot(  
      index = "anno",  
      columns = "ambito_nome",  
      values = "totale"  
    )  
  
totale_gruppo
```



ambito_nome	Abbiategrasso	Argonne	Bisceglie	Bocconi	Buenos Aires - Bacone	Centro Direzionale	Centro storico	Farini	Ferroviaria Nord-Ovest	Fiera	...
anno											
2016	NaN	1.0	NaN	1.0	NaN	1.0	6.0	NaN	NaN	3.0	...
2018	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1.0	...
2019	NaN	NaN	2.0	NaN	1.0	2.0	3.0	NaN	NaN	NaN	...
2020	1.0	1.0	NaN	2.0	NaN	1.0	1.0	NaN	NaN	5.0	...
2021	1.0	NaN	NaN	1.0	2.0	3.0	6.0	1.0	1.0	2.0	...
2022	6.0	7.0	4.0	NaN	2.0	4.0	9.0	NaN	1.0	14.0	...

6 rows × 32 columns

Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

Le zone, però, sono proprio tante; per ridurle decidiamo di visualizzare solo quelle che hanno un minimo di 8 colonnine.

```
✓ [12] totale_gruppo = totale_gruppo[totale_nome_min_8.index]
```

```
totale_gruppo
```

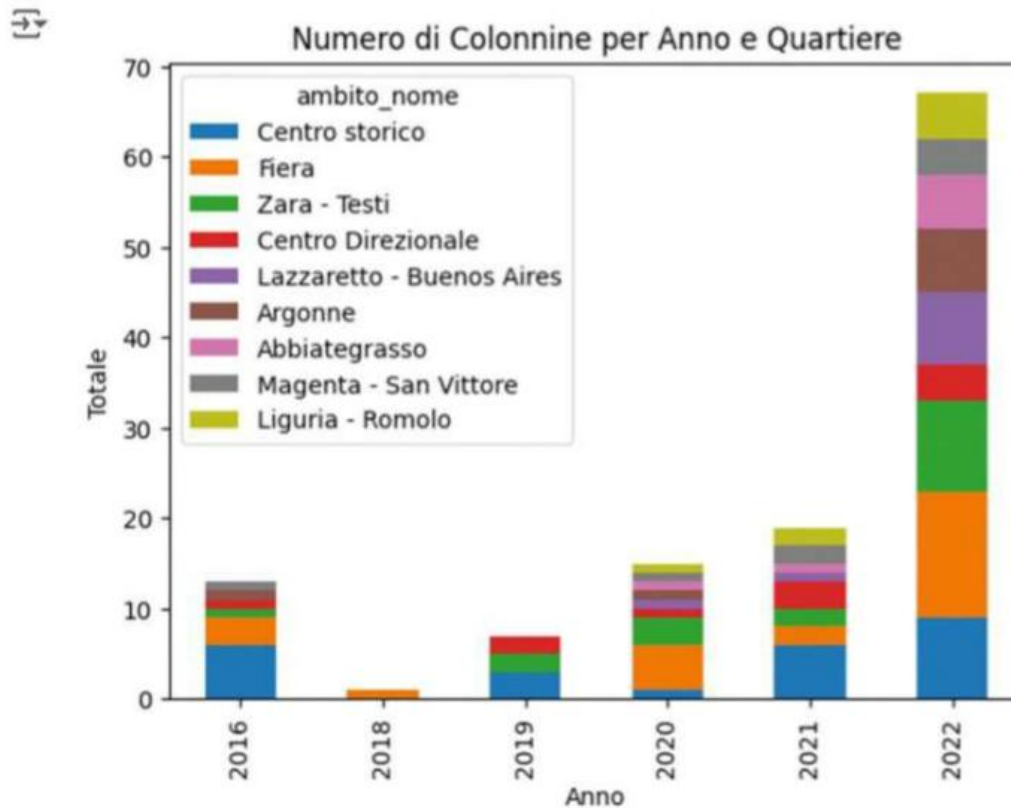


ambito_nome	Centro storico	Fiera	Zara - Testi	Centro Direzionale	Lazzaretto - Buenos Aires	Argonne
anno						
2016	6.0	3.0	1.0	1.0	NaN	1.0
2018	NaN	1.0	NaN	NaN	NaN	NaN
2019	3.0	NaN	2.0	2.0	NaN	NaN
2020	1.0	5.0	3.0	1.0	1.0	1.0
2021	6.0	2.0	2.0	3.0	1.0	NaN
2022	9.0	14.0	10.0	4.0	8.0	7.0

Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

Una visualizzazione molto efficace in questo caso è data da un grafico a barre in cui le zone sono «impilate» (in inglese *stacked*) una sull'altra.

```
[13] _ = totale_gruppo.plot(  
    title = "Numero di Colonnine per Anno e Quartiere",  
    xlabel = "Anno",  
    ylabel = "Totale",  
    kind = "bar",  
    stacked = True  
)
```



Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

7 Visualizziamo in una cartina la posizione delle colonnine di ricarica.

Le coordinate GPS sono salvate come stringa. Di fatto, ogni stringa è una tupla di due numeri in virgola mobile che rappresentano, rispettivamente, latitudine e longitudine. Dato che le stringhe della colonna `Location` rispettano la sintassi Python delle tuple, possiamo applicare la funzione `eval`, che valuta la stringa come se fosse codice Python. Questa funzione chiede all'interprete

Python di eseguire quella stringa come se fosse codice, convertendo di fatto la stringa in una tupla di due valori.

Per visualizzare le coordinate su un piano cartesiano ci serve poter separare la colonna `Location` in due colonne, una per i valori dell'asse `x` e una per i valori dell'asse `y`. Il modo più semplice è quello di convertire la colonna in una lista di tuple con il metodo `tolist`, per poi assegnare la lista a due nuove colonne nel `DataFrame`: ci penserà *Pandas* a dividere i valori nelle tuple tra le due colonne.

```
✓ [14] data[["Latitudine", "Longitudine"]] = data["Location"].apply(eval).tolist()  
0s  
data[["Latitudine", "Longitudine"]].head()
```



	Latitudine	Longitudine
0	45.462149	9.193577
1	45.467629	9.177604
2	45.484484	9.204908
3	45.484277	9.235418
4	45.437302	9.169170



Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

Le coordinate GPS vanno visualizzate in un sistema di riferimento, quindi è necessario poter disegnare anche una mappa del territorio di Milano. Per farlo ricorreremo di nuovo agli Open Data del Comune di Milano, questa volta nella variante geospaziale. La mappa dei municipi di Milano è salvata in un archivio ZIP; ci basterà scaricare il file, passare il contenuto della richiesta al gestore degli archivi ZIP ed estrarre il file che contiene i poligoni da disegnare.

```
✓ [15] # URL mappa compressa.  
15 mappa = "".join([  
    "https://geoportale.comune.milano.it/Download/area_download/",  
    "SIT/Municipi/Municipi_RDN2008.zip"  
])  
# Scaricamento mappa compressa.  
mappa = requests.get(mappa)  
# Decompressione mappa.  
mappa = zipfile.ZipFile(io.BytesIO(mappa.content))  
mappa.extractall(".")  
# Lettura mappa.  
mappa = gpd.read_file("Municipi.shp")  
# Conversione a punti (Latitudine, Longitudine).  
mappa = mappa.to_crs(crs = 4326)  
  
mappa
```

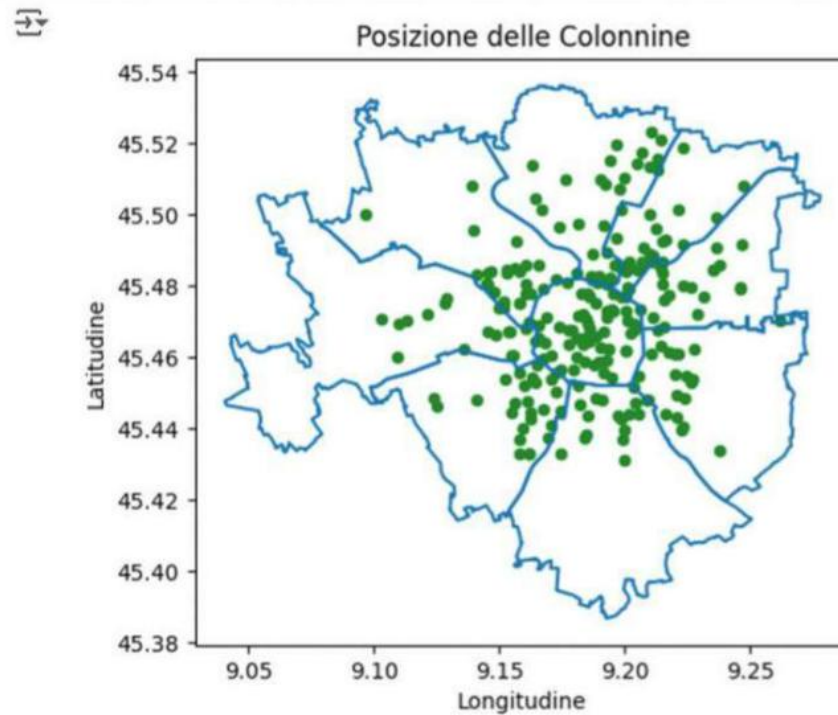
	MUNICIPIO	AREA	PERIMETRO	geometry
0	1	9.426872e+06	11935.5825	POLYGON ((9.19441 45.47949, 9.19563 45.47896, ...
1	8	2.390648e+07	32807.3751	POLYGON ((9.09930 45.53128, 9.09933 45.53118, ...
2	9	2.101349e+07	27157.1251	POLYGON ((9.17572 45.53560, 9.17574 45.53537, ...
3	2	1.262855e+07	20971.2966	POLYGON ((9.22393 45.52357, 9.22395 45.52348, ...
4	3	1.443424e+07	25665.7875	POLYGON ((9.26874 45.51035, 9.26875 45.51035, ...
5	4	2.069548e+07	22716.0101	POLYGON ((9.26759 45.47204, 9.26781 45.47140, ...
6	7	3.136342e+07	45390.1230	POLYGON ((9.07310 45.50604, 9.07327 45.50589, ...
7	6	1.833622e+07	22297.3464	POLYGON ((9.13975 45.46092, 9.14163 45.46061, ...
8	5	2.995886e+07	29712.8420	POLYGON ((9.18957 45.45197, 9.19036 45.45195, ...

Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

È necessario infine fare un ultimo passaggio, per ricondurre le coordinate utilizzate dal sistema di riferimento italiano a quelle del sistema GPS, così da ottenere coppie di valori di latitudine e longitudine coerenti con la posizione delle colonnine.

Ora non ci rimane che visualizzare il tutto sulla stessa figura.

```
✓ [16] # Crea una nuova figura.  
0s: fig, ax = plt.subplots()  
# Visualizza contorni della mappa.  
_ = mappa.boundary.plot(ax = ax)  
# Aggiungi colonnine.  
_ = data.plot(  
    title = "Posizione delle Colonnine",  
    x = "Longitudine",  
    y = "Latitudine",  
    kind = "scatter",  
    color = "green",  
    ax = ax  
)
```



Esercitazione guidata – La ricarica dei veicoli elettrici a Milano

Per realizzare nuove visualizzazioni potete provare a modificare il notebook, magari utilizzando la colonna «tipologia», che abbiamo selezionato ma non abbiamo utilizzato all'interno dell'esercitazione.

Buon divertimento!

