

Python **GIORNO 3 - Librerie standard, librerie esterne, NumPy, Pandas**

Agenda:

- Librerie
- NumPy
- Introduzione a Pandas



Perché?

- Uno dei punti di forza di Python è la quantità di **librerie**, interne ed esterne. Queste librerie contengono funzioni, classi e metodi che ci permettono di sfruttare al massimo il potenziale di Python **senza dover "reinventare la ruota"**, ma utilizzando il lavoro che altri hanno messo a disposizione.
- Pandas, che è basato su NumPy, è la libreria che come Data Analyst vi troverete ad usare **quotidianamente**.

Alla fine di questa lezione saprete:

- Cos'è una libreria
- Qual è la differenza tra una libreria interna ed una esterna
- Come usarle nel nostro codice
- Come installare le librerie esterne
- Cos'è NumPy e il suo datatype base: gli `ndarray`
- Cos'è Pandas, quali sono i suoi datatype base (`Series` e `DataFrame`)
- Come collegarci tramite Python ad un database per leggerne i dati
- Come fare le prime interrogazioni su un dataframe

Libraries

Una libreria è un insieme di moduli e funzioni predefinite che sono stati scritti per svolgere specifiche operazioni o per fornire funzionalità aggiuntive al linguaggio di programmazione.



Libraries

Le librerie sono quindi dei pezzi di codice che qualcuno ha scritto (potremmo anche essere stati noi), e che possiamo usare nel nostro codice. Questo è possibile grazie alla struttura **open-source di Python**: il suo codice sorgente è pubblico e tutti possono contribuire al suo sviluppo.

(Siete curiosi? Qui: <https://github.com/python/cpython/tree/main> c'è Python, riga per riga).

Alcune librerie sono già incluse nel momento in cui installiamo Python e fanno parte di quella che viene chiamata **Standard Library**: sono scritte e mantenute direttamente da Python.

Esistono poi delle **librerie esterne**, che vanno invece installate separatamente e che sono invece scritte e mantenute dalla community degli utenti di Python.

Standard Library

Nella Standard Library sono incluse quelle funzioni built-in che abbiamo visto (come `print()` e `help()`), che sono disponibili direttamente nei nostri notebook, così come i data types base.

Per avere accesso invece alle librerie incluse nella Standard Library abbiamo bisogno di utilizzare il comando `import`

Esistono librerie, moduli e pacchetti che si occupano di interagire con il sistema operativo, che integrano calcoli matematici, che interagiscono con determinati tipi di files, che permettono di manipolare le date, e ancora e ancora. Troviamo un elenco esaustivo nella documentazione:
<https://docs.python.org/3/library/>

Standard Library

Ad esempio, esiste la libreria `math`, che contiene funzioni matematiche.

<https://docs.python.org/3/library/math.html>

Abbiamo visto che Python non ha un'operatore built-in per calcolare la `radice quadrata` di un numero, ma la libreria `math` include la funzione `sqrt()`.

Dalla documentazione:

```
math.sqrt(x)
Return the square root of x.
```

Standard Library

Se però proviamo ad utilizzarla in modo diretto nel nostro codice, otteniamo un errore:

```
1 math.sqrt(64)

-----
NameError          Traceback (most recent call last)
<ipython-input-1-226f3d47e899> in <cell line: 1>()
----> 1 math.sqrt(64)

NameError: name 'math' is not defined
```

Questo perché Python non carica tutte le sue librerie di default, per questioni di prestazioni e velocità. Per poter aver accesso ad una libreria e alle sue funzioni, dobbiamo usare il comando **import**.

```
1 import math
2
3 math.sqrt(64)
```

8.0

Standard Library

Un elenco non esaustivo di librerie utili incluse nella Standard Library può essere:

- **os**: gestisce le interazioni con il sistema operativo

```
1 import os  
2 os.path.isfile("ma_questo_file_esiste_o_no.txt")  
  
False
```

- **datetime**: gestisce e permette operazioni sulle date e sul tempo

```
1 import datetime  
2 datetime.datetime.now().date()  
  
datetime.date(2024, 3, 7)
```

- **random**: gestisce delle operazioni pseudo-random

```
1 import random  
2 my_list = ["tanti", "elementi", "nella", "mia", "lista"]  
3 random.choice(my_list)  
  
'mia'
```

Librerie esterne

La community di Python è vitale, popolosa ed attiva: ci sono molti sviluppatori che scrivono delle librerie e le mettono **a disposizione del pubblico gratuitamente.**

State cercando una libreria per lavorare su dati geospaziali? C'è [GeoPandas](#)

Volete lavorare con le immagini e la computer-vision? C'è [OpenCV](#)

Avete bisogno di gestire il backend di un API o di una pagina web? C'è [FastAPI](#)

Quasi tutte le librerie esterne sono disponibili al **Python Package Index:** <https://pypi.org/>

Due librerie esterne fondamentali per l'analisi dei dati sono [NumPy](#) e [Pandas](#).

Librerie esterne

Le librerie esterne, che dunque non sono incluse nell'installazione base di Python, hanno bisogno di essere installate a parte.

Se state usando JupyterLab tramite Anaconda avrete già Numpy e Pandas disponibili, perché Anaconda li installa di default.

Se invece, provando a digitare `import numpy`, riceviamo un errore di tipo `ModuleNotFoundError`

```
>>> import numpy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
```

Abbiamo bisogno di installarlo.

Librerie esterne

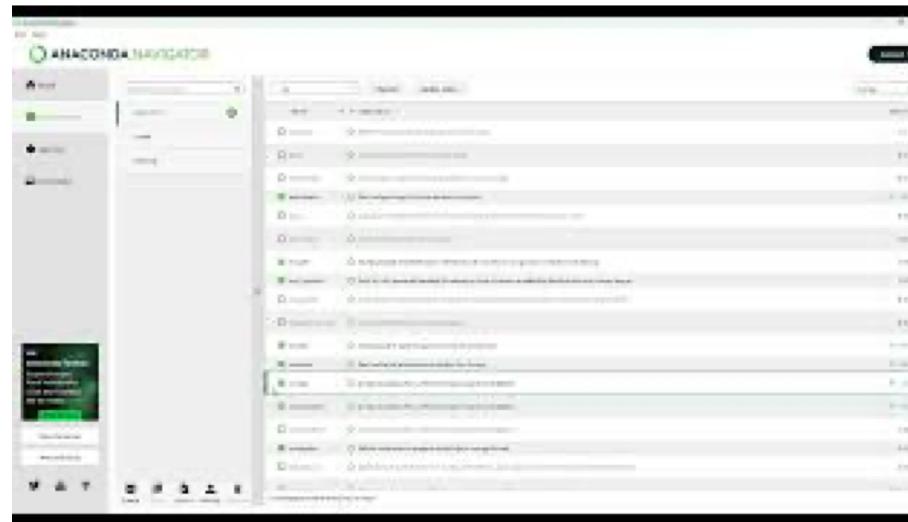
Per installare delle librerie esterne utilizziamo il comando **pip**, che non è una istruzione Python, ma un **comando da terminale**

Possiamo quindi eseguirlo:

- da **terminale**, digitiamo `pip install numpy` e premiamo invio
- da un **Jupyter Notebook**, in una nuova cella di codice digitiamo `pip install numpy` ed eseguiamo la cella; anche se non è codice Python, Jupyter è in grado di eseguirlo come se lo stessimo facendo appunto da terminale

Librerie esterne

Possiamo anche installare delle librerie utilizzando Anaconda nella sezione Environments, selezionabile dal menù a sinistra.



NumPy

NumPy (abbreviazione di Numerical Python) è un pacchetto Python per lavorare con array multidimensionali, che quindi supera le limitazioni dei tipi di dato nativi, che sono monodimensionali.



NumPy

Le caratteristiche fondamentali di NumPy sono:

- **Array multidimensionali:** NumPy fornisce un nuovo tipo di dato avanzato chiamato `ndarray`, che è estremamente efficiente per memorizzare e manipolare dati in array multidimensionali.
- **Operazioni matematiche:** NumPy offre un'ampia gamma di funzioni matematiche, tra cui funzioni trigonometriche, logaritmiche, esponenziali e altre funzioni matematiche speciali
- **Indicizzazione avanzata:** NumPy offre una vasta gamma di tecniche per indicizzare gli array, che consentono di selezionare e manipolare porzioni specifiche degli array NumPy in modo flessibile ed efficiente
- **Integrazione con altre librerie:** NumPy è ampiamente utilizzato come base per molte altre librerie scientifiche e di calcolo in Python, come ad esempio SciPy, `Pandas` e Scikit-Learn

Tutto questo è implementato tramite delle strutture ottimizzate veloci e performanti.

NumPy

Proviamo, per esempio, ad importare NumPy e a trasformare una normale lista in un ndarray.

```
1 import numpy as np          # diamo un alias all'import con la keyword "as"
2 my_list = [1, 2, 3, 4, 5]
3 my_array = np.array(my_list) # casting
4 type(my_array)
```

Alcuni dei **metodi** degli ndarray sono:

```
1 numeri = [1, 5, 20, 18, 50, 4, -7, 12, 25, 10]
2 numery = np.array(numeri)
3 print(type(numery))
4 print(numery.min())
5 print(numery.max())
6 print(numery.mean())
7 print(numery.argmax())

<class 'numpy.ndarray'>
-7
50
13.8
4
```

NumPy

Una delle caratteristiche introdotte da NumPy è il **masking**, ovvero la possibilità di effettuare operazioni su tutti gli elementi in una sola volta.

In pratica, il masking consiste nell'effettuare un'operazione (moltiplicazione, divisione, espressione booleana, eccetera) su un array "mascherato" come se fosse un singolo valore: in realtà l'operazione viene effettuata su tutti gli elementi dell'array, cosa non possibile con le liste:

Trasformando questa lista in un `ndarray` otteniamo un risultato diverso:

```
1 numeri = [1, 5, 20, 18, 50, 4, -7, 12, 25, 10]
2 print(numeri / 3)
3

-----
TypeError                                         Traceback (most recent call last)
<ipython-input-27-a5976c1c6598> in <cell line: 2>()
      1 numeri = [1, 5, 20, 18, 50, 4, -7, 12, 25, 10]
----> 2 print(numeri / 3)

TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

```
1 numeri = [1, 5, 20, 18, 50, 4, -7, 12, 25, 10]
2 numery = np.array(numeri)
3 numery / 3

array([ 0.33333333,  1.66666667,  6.66666667,  6.          ,
       16.66666667,
      1.33333333, -2.33333333,  4.          ,  8.33333333,  3.33333333])
```

NumPy

E se passassimo una condizione, che dovrebbe risultare in un booleano?

```
1 numeri = [1, 5, 20, 18, 50, 4, -7, 12, 25, 10]
2 numery = np.array(numeri)
3 numery > 20

array([False, False, False, False, True, False, False, True,
       False])
```

Grazie al masking, noi possiamo specificare dei veri e propri **filtr che definiscono delle condizioni di estrazione** (o, in effetti, delle *query*)

```
1 numeri = [1, 5, 20, 18, 50, 4, -7, 12, 25, 10]
2 numery = np.array(numeri)
3 my_filter = numery > 20
4 numery[my_filter]

array([50, 25])
```

Pandas

Pandas è una potente libreria open-source per la manipolazione e l'analisi dei dati basata su NumPy. Offre strutture dati flessibili e performanti, che consentono di lavorare con dati strutturati in modo intuitivo e efficace.



Pandas

Le caratteristiche fondamentali di Pandas sono:

- **DataFrame e Series:** Due nuovi datatype: il `DataFrame`, una tabella bidimensionale simile a un foglio di calcolo, e la `Series`, un array unidimensionale; entrambi sono etichettati.
- **Indicizzazione:** Pandas offre un'indicizzazione versatile che permette di selezionare, filtrare e modificare i dati in base a criteri specifici utilizzando sia le etichette delle righe e delle colonne che gli indici numerici per accedere ai dati.
- **Gestione dei dati mancanti:** Pandas fornisce strumenti per gestire i dati mancanti in modo efficace, consentendo agli utenti di riempire, rimuovere o interpolare i valori mancanti all'interno dei `DataFrame`.
- **Operazioni di gruppo e aggregazione:** La libreria supporta operazioni di gruppo e aggregazione sui dati, permettendo agli utenti di suddividere i dati in base a criteri specifici, eseguire operazioni su ciascun gruppo e combinare i risultati.

pd.Series

Il tipo di dato `Series` è un **array monodimensionale indicizzato**, il cui indice può essere sia numerico che categorico (cioè rappresentato da una stringa).

Contiene quindi sia dei dati, sia dei metadati.

Creiamo la nostra prima `Series` di esempio:

```
1 import pandas as pd
2 pd.Series([0.25, 0.5, 0.75, 1, 1.5, 2])

0    0.25
1    0.50
2    0.75
3    1.00
4    1.50
5    2.00
dtype: float64
```

Possiamo vedere che la nostra lista è stata convertita in un datatype diverso che presenta degli **indici** a sinistra (riquadro arancio) e gli **elementi** (*i.e.*, i dati) della nostra lista a destra (riquadro azzurro).

Di default, Pandas assegna ad ogni elemento un indice numerico ed incrementale partendo da 0.

pd.Series

Il tipo di dato `Series` è un **array monodimensionale indicizzato**, il cui indice può essere sia numerico che categorico (cioè rappresentato da una stringa).

Contiene quindi sia dei dati, sia dei metadati.

Creiamo la nostra prima `Series` di esempio:

```
1 import pandas as pd  
2 pd.Series([0.25, 0.5, 0.75, 1, 1.5, 2])
```

0	0.25
1	0.50
2	0.75
3	1.00
4	1.50
5	2.00

dtype: float64

Possiamo vedere che la nostra lista è stata convertita in un datatype diverso che presenta degli **indici** a sinistra (riquadro arancio) e gli **elementi** (*i.e.*, i dati) della nostra lista a destra (riquadro azzurro).

Di default, Pandas assegna ad ogni elemento un indice numerico ed incrementale partendo da 0.

pd.Series

Abbiamo anche la possibilità di scegliere degli indici ed esplicitarli durante la definizione della Series:

```
1 import pandas as pd
2 pd.Series([0.25, 0.5, 0.75, 1, 1.5, 2],
3            index=list("abcdef"))

a    0.25
b    0.50
c    0.75
d    1.00
e    1.50
f    2.00
dtype: float64
```

Gli indici possono essere non-univoci, è importante però che forniamo una quantità di indici uguale alla quantità di valori della Series.

pd.Series

Pandas si integra molto bene anche con i dizionari. Proviamo a creare una `Series` partendo da un `dict`:

```
1 import pandas as pd
2 my_dict = {"a": 1, "b": 1.5, "c": 2, "d": 2.5}
3 pd.Series(my_dict)

a    1.0
b    1.5
c    2.0
d    2.5
dtype: float64
```

La `Series` che otteniamo ha come indici le chiavi del nostro dizionario, e come valori, appunto, i valori del dizionario.

pd.Series

Per accedere ai dati possiamo usare l'indicizzazione con parentesi quadre.

Possiamo usare gli indici numerici:

```
1 import pandas as pd
2 my_series = pd.Series([0.25, 0.5, 0.75, 1, 1.5, 2])
3 my_series[1]
0.5
```

Gli indici categorici:

```
1 import pandas as pd
2 indexed = pd.Series([0.25, 0.5, 0.75, 1, 1.5, 2],
3                      index=list("abcdef"))
4 indexed["a"]
0.25
```

pd.Series

ATTENZIONE! Se gli indici della nostra Series sono degli int, l'indice che inseriamo tra parentesi quadre, in questo caso, non si riferisce alla posizione dell'elemento all'interno della Series, ma all'indice della stessa!

Se assegniamo alla Series degli indici “disordinati” vediamo come il riferimento non è alla posizione del dato ma al “nome” dell'indice:

```
1 import pandas as pd
2 indexed = pd.Series([0.25, 0.5, 0.75, 1, 1.5, 2],
3                      index=[0, 2, 4, 3, 1, 6])
4 indexed
0    0.25
2    0.50
4    0.75
3    1.00
1    1.50
6    2.00
dtype: float64
1 indexed[1]
1.5
```

pd.Series

Dato che gli indici possono essere ripetuti, se indicizziamo una Series tramite un indice ripetuto non avremo un singolo valore di ritorno ma un'altra Series contenente tutti quei valori che corrispondono a quell'indice, e l'indice stesso:

```
1 import pandas as pd
2 indexed = pd.Series([0.25, 0.5, 0.75, 1, 1.5, 2],
3                      index=list("aaabcd"))
4 indexed["a"]

a    0.25
a    0.50
a    0.75
dtype: float64
```

pd.Series

È possibile accedere ai dati tramite l'operatore **due punti**:

```
1 import pandas as pd
2 my_series = pd.Series([0.25, 0.5, 0.75, 1, 1.5, 2],
3                      index=list("abcdef"))
4 my_series["b":"d"]

b    0.50
c    0.75
d    1.00
dtype: float64
```

L'ultimo elemento è compreso; la logica è di andare da un'etichetta all'altra: ergo, non si tratta di slicing ma di **un range da un'etichetta a un'altra**.

pd.DataFrame

Il tipo di dato `DataFrame` è una **struttura dati tabellare bidimensionale** che può memorizzare differenti tipi di dato e in cui ogni asse è etichettato.

Si può creare a partire da un tipo di dato bidimensionale (ad esempio una matrice `ndarray` di NumPy) oppure da un dizionario di liste, dove le chiavi sono i metadati, e le liste i dati delle diverse colonne

```
1 import pandas as pd
2 data = {"cybersecurity": [10, 20, 30],
3          "data_analyst": [40, 50, 60]}
4 studenti = pd.DataFrame(data, index=["anno1", "anno2", "anno3"])
5 studenti
```

	cybersecurity	data_analyst	
anno1	10	40	lib
anno2	20	50	
anno3	30	60	

Jupyter e Colaboratory implementano delle visualizzazioni avanzate dei `DataFrame`, come si vede nello screenshot qui sopra.

pd.DataFrame

Hanno due livelli di indicizzazione: quello degli **indici**, come per le `Series`, a cui si aggiunge quello delle **colonne**.

```
1 import pandas as pd
2 data = {"cybersecurity": [10, 20, 30],
3          "data_analyst": [40, 50, 60]}
4 studenti = pd.DataFrame(data, index=["anno1", "anno2", "anno3"])
5 studenti
```

	cybersecurity	data_analyst
anno1	10	40
anno2	20	50
anno3	30	60

Possiamo pensarlo come a molte `Series` che condividono lo stesso indice, ognuna con il suo titolo, oppure come a una tabella.

pd.DataFrame

Un altro esempio di DataFrame:

```
1 import pandas as pd
2 popolazione = {"Roma": 2_748_109, "Milano": 1_354_196,
3     "Napoli": 913_462, "Torino": 841_600,
4     "Palermo": 630_167, "Genova": 558_745}
5 superficie = {"Roma": 1_287.24, "Milano": 181.68,
6     "Napoli": 118.94, "Torino": 130.06,
7     "Palermo": 160.59, "Genova": 240.29}
8 dati_citta = pd.DataFrame({"popolazione": popolazione,
9                             "superficie": superficie})
10 dati_citta
```

	popolazione	superficie	
Roma	2748109	1287.24	
Milano	1354196	181.68	
Napoli	913462	118.94	
Torino	841600	130.06	
Palermo	630167	160.59	
Genova	558745	240.29	

pd.DataFrame

Il tipo di dato `DataFrame` è dunque già predisposto per lavorare con tabelle bidimensionali etichettate (cioè con metadati) come per Excel, SQL o file di tipo JSON o CSV.

Possiamo infatti importare facilmente dei dati da un file CSV, o da un database, grazie a Pandas.

Importare dati da un database esterno

Per importare dei dati da un database esterno, nel nostro caso un database MySQL, abbiamo bisogno di installare alcune librerie che non sono incluse nella Standard Library.

Utilizzando uno dei metodi già visti andiamo ad installare:

- **PyMySQL**: un driver che permette di connettersi a MySQL tramite Python.
`pip install pymysql`
- **SQLAlchemy**: una libreria che permette di gestire le query ad un database.
`pip install SQLAlchemy`
- **Dotenv**: una libreria che useremo per gestire le credenziali di accesso al database in modo sicuro.
`pip install dotenv`

Importare dati da un database esterno

Una volta installate queste librerie, creiamo, nella stessa cartella dove abbiamo il nostro notebook, un file chiamato semplicemente “`.env`” e andiamo a scriverci dentro quanto segue:

```
username=<YOURUSERNAME>
password=<YOURPASSWORD>
host=<YOURHOST>
dbname=<YOURDBNAME>
```

Sostituite i valori dopo gli uguali (=) con quelli forniti dal vostro insegnante.

Nel vostro Notebook, poi, eseguite:

```
1 import dotenv
2 dotenv.load_dotenv()
```

Importare dati da un database esterno

Perché creare un file `.env` e non scrivere direttamente le credenziali di accesso al database all'interno del notebook?

Perché nessun tipo di credenziale va MAI inserita in chiaro all'interno di un file. Questa è una *best practice* di sicurezza fondamentale che non dobbiamo ignorare.

Un file senza nome e che ha solo l'estensione, come in questo caso, viene tipicamente nascosto dai sistemi operativi e non è immediatamente visibile, a meno di non attivare intenzionalmente l'opzione di visualizzazione dei file nascosti.

Non condivideremo MAI il nostro file `.env`, quindi le nostre credenziali saranno sempre al sicuro. Se utilizzate GitHub, assicuratevi di aggiungere al file `.gitignore` della vostra repository una riga con scritto `**/.env`, così sarete sicuri che non uscirà dal vostro computer.

Importare dati da un database esterno

Una volta fatto questo, andiamo a recuperare i valori dalle variabili d'ambiente (questo concetto può sembrare un po' oscuro al momento, ma fidatevi del processo e diventerà più chiaro in futuro) e stabiliamo una connessione al nostro database.

```
import os
import dotenv
import sqlalchemy
import pandas as pd

dotenv.load_dotenv(override=True, dotenv_path=<PATH AL FILE>)

username = os.getenv("username")
password = os.getenv("password")
host = os.getenv("host")
dbname = os.getenv("dbname")

connection_string = "mysql+pymysql://" + username + ":" + password + "@" + host + "/" + dbname

db_engine = sqlalchemy.create_engine(connection_string)
```

Importare dati da un database esterno

Definiamo una query esattamente come faremmo in un DBMS, inseriamola in una stringa e usiamo il metodo `.read_sql()` di Pandas per caricare i dati risultanti dalla query in un database.

```
query = "SELECT * FROM dimproduct"

tabella = pd.read_sql(query, db_engine)

print(tabella)
```

ProductKey	ProductAlternateKey	ProductSubcategoryKey	WeightUnitMeasureCode	SizeUnitMeasureCode	EnglishProductName	SpanishProductName	FrenchProductName	StandardCost	FinishedGoodsFlag
0	1	AR-5381	NaN	None	None	Adjustable Race			NaN	0	...
1	2	BA-8327	NaN	None	None	Bearing Ball			NaN	0	...
2	3	BE-2349	NaN	None	None	BB Ball Bearing			NaN	0	...
3	4	BE-2908	NaN	None	None	Headset Ball Bearings			NaN	0	...
4	5	BL-2036	NaN	None	None	Blade			NaN	0	...
...
601	602	BB-8107	5.0	G	None	ML Bottom Bracket	Eje de pedalier GM	Axe de pédalier ML	44.95	1	...
602	603	BB-9108	5.0	G	None	HL Bottom Bracket	Eje de pedalier GA	Axe de pédalier HL	53.94	1	...
603	604	BK-R19B-44	2.0	LB	CM	Road-750 Black, 44	Carretera: 750, negra, 44	Vélo de route 750 noir, 44	343.65	1	...
604	605	BK-R19B-48	2.0	LB	CM	Road-750 Black, 48	Carretera: 750, negra, 48	Vélo de route 750 noir, 48	343.65	1	...
605	606	BK-R19B-52	2.0	LB	CM	Road-750 Black, 52	Carretera: 750, negra, 52	Vélo de route 750 noir, 52	343.65	1	...



GRAZIE
EPCODE