Laboratorio di programmazione

Python

A.A. 2020-2021

Lezione 9

Approfondimento: `import`

Abbiamo già visto che `import` è la *keyword* con cui importiamo nel nostro programma istruzioni contenute in un altro file.

Più correttamente: la *keyword* `import` serve a trovare e caricare **moduli** (singolo file `.py`) o **pacchetti/sotto-pacchetti** (insieme di file `.py`).

Approfondimento: `import`

Abbiamo già visto che `import` è la *keyword* con cui importiamo nel nostro programma istruzioni contenute in un altro file.

Più correttamente: la *keyword* `import` serve a trovare e caricare **moduli** (singolo file `.py`) o **pacchetti/sotto-pacchetti** (insieme di file `.py`).

In [1]:
```
import turtle
print(type(turtle))
```

```
<class 'module'>
```

Approfondimento: `import`

Abbiamo già visto che `import` è la *keyword* con cui importiamo nel nostro programma istruzioni contenute in un altro file.

Più correttamente: la *keyword* `import` serve a trovare e caricare **moduli** (singolo file `.py`) o **pacchetti/sotto-pacchetti** (insieme di file `.py`).

In [1]:
```
import turtle
print(type(turtle))
```

```
<class 'module'>
```

Il modulo che importiamo deve esistere nel sistema dove stiamo lavorando, ovvero deve esistere nel vostro computer il file in cui sono contenute le classi e le funzioni specifiche di quel modulo:

Approfondimento: `import`

Abbiamo già visto che `import` è la *keyword* con cui importiamo nel nostro programma istruzioni contenute in un altro file.

Più correttamente: la *keyword* `import` serve a trovare e caricare **moduli** (singolo file `.py`) o **pacchetti/sotto-pacchetti** (insieme di file `.py`).

In [1]:
```
import turtle
print(type(turtle))
```

```
<class 'module'>
```

Il modulo che importiamo deve esistere nel sistema dove stiamo lavorando, ovvero deve esistere nel vostro computer il file in cui sono contenute le classi e le funzioni specifiche di quel modulo:

In [2]:
```
import modulo_che_non_esiste
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-2-54f6f59ce10f> in <module>
----> 1 import modulo_che_non_esiste

ModuleNotFoundError: No module named 'modulo_che_non_esiste'
```

`import` e *namespace*

Il **modulo** che importiamo attraverso la *keyword* `import` crea un suo *namespace*. Classi e funzioni del modulo esistono solo all'interno di quel namespace.

`import` e *namespace*

Il **modulo** che importiamo attraverso la *keyword* `import` crea un suo *namespace*. Classi e funzioni del modulo esistono solo all'interno di quel namespace.

In [3]:
```
import turtle
raffaello = turtle.Turtle()  # Turtle esiste nel namespace turtle
```

`import` e *namespace*

Il **modulo** che importiamo attraverso la *keyword* `import` crea un suo *namespace*. Classi e funzioni del modulo esistono solo all'interno di quel namespace.

In [3]:
```
import turtle
raffaello = turtle.Turtle()  # Turtle esiste nel namespace turtle
```

In [4]:
```
donatello = Turtle()  # Turtle NON esiste nel global namespace
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-414ca8ad738c> in <module>
----> 1 donatello = Turtle()

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

In Python è possibile ridefinire il *namespace* del modulo che stiamo importando, creando un **alias** (generalmente più breve ed immediato da utilizzare) da utilizzare all'interno del programma:

In Python è possibile ridefinire il *namespace* del modulo che stiamo importando, creando un **alias** (generalmente più breve ed immediato da utilizzare) da utilizzare all'interno del programma:

In [5]:
```
import turtle as t
raffaello = t.Turtle()
```

In Python è possibile ridefinire il *namespace* del modulo che stiamo importando, creando un **alias** (generalmente più breve ed immediato da utilizzare) da utilizzare all'interno del programma:

In [5]:
```
import turtle as t
raffaello = t.Turtle()
```

> *Alcuni moduli sono talmente famosi ed utilizzati che vengono importati ridefinedno il loro namespace per convenzione:*
>
> ```
> import numpy as np
> import pandas as pd
> import matplotlib.pyplot as plt
> ```

Importare singole classi/funzioni

Non siamo obbligati ad importare l'intero modulo, ma possiamo imporare solo la classe o la funzione, contenuta in quel modulo, che ci interessa:

Importare singole classi/funzioni

Non siamo obbligati ad importare l'intero modulo, ma possiamo imporare solo la classe o la funzione, contenuta in quel modulo, che ci interessa:

In [6]:
```
from turtle import Turtle
raffaello = Turtle()
```

Importare singole classi/funzioni

Non siamo obbligati ad importare l'intero modulo, ma possiamo imporare solo la classe o la funzione, contenuta in quel modulo, che ci interessa:

In [6]:
```
from turtle import Turtle
raffaello = Turtle()
```

Una volta importata, la classe `Turtle` vive nel *global namespace* e possiamo utilizzarla senza specificare il *namespace* del modulo `turtle`.

> ***Attenzione**: come abbiamo visto in precedenza, specificare i namespace evita ambiguità. Bisogna sempre fare attenzione a come importiamo classi e funzioni esterne nel nostro programma.*

NON FATE MAI `from module import *`

Python permette di importare **tutto** il contenuto di un modulo con il *token* "generico" `*`.

Questa può sembrare un'idea "comoda", ma fa perdere (a noi e all'interprete) l'informazione sul namespace del modulo che stiamo usando...e questa **NON** è mai una buona idea!

Vediamolo con un esempio:

NON FATE MAI `from module import *`

Python permette di importare **tutto** il contenuto di un modulo con il *token* "generico" `*`.

Questa può sembrare un'idea "comoda", ma fa perdere (a noi e all'interprete) l'informazione sul namespace del modulo che stiamo usando...e questa **NON** è mai una buona idea!

Vediamolo con un esempio:

In [9]:
```
import numpy
import math

print(math.sqrt(4))
print(numpy.sqrt([4,9,25]))
```

```
2.0
[2. 3. 5.]
```

Vediamo che succede a fare i "pigri" e a importare con `*`:

Vediamo che succede a fare i "pigri" e a importare con `*`:

In [10]:
```
from numpy import *
from math import *

print(sqrt(4))
print(sqrt([4,9,25]))
```

```
2.0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-37336a4edcd6> in <module>
      3
      4 print(sqrt(4))
----> 5 print(sqrt([4,9,25]))

TypeError: must be real number, not list
```

Vediamo che succede a fare i "pigri" e a importare con `*`:

In [10]:
```
from numpy import *
from math import *

print(sqrt(4))
print(sqrt([4,9,25]))
```

```
2.0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-37336a4edcd6> in <module>
      3
      4 print(sqrt(4))
----> 5 print(sqrt([4,9,25]))

TypeError: must be real number, not list
```

> *Il secondo import sovrascrive le funzioni con lo stesso nome. il codice in esempio non darebbe problemi invertendo gli import. Per andare sul sicuro è sempre bene importare esplicitamente i namespace e in generale:*
>
> ***NON FATE MAI:***
>
> ```
> from module import *
> ```

Importare codice scritto da noi

Non siamo obbligati a importare solo dai moduli che vengono forniti direttamente installando Python (`this`, `turtle`, `math`) o da moduli aggiuntivi ufficiali (`numpy`, `pandas`, `matplotlib`).

Possiamo anche importare i **nostri** moduli:

```
import mio_modulo

import mio_modulo as mm

from mio_modulo import MyClass
```

`numpy`

Che cos'è `numpy` e perchè è così famoso:

> *NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.*

In pratica, `numpy` è un pacchetto Python che consente di lavorare con vettori:

`numpy`

Che cos'è `numpy` e perchè è così famoso:

> *NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.*

In pratica, `numpy` è un pacchetto Python che consente di lavorare con vettori:

In [12]:
```
vettore = np.array([1,2,4,3,5])
print(vettore)
print(type(vettore))
```

```
[1 2 4 3 5]
<class 'numpy.ndarray'>
```

E matrici:

E matrici:

In [14]:
```
matrice = np.array([[0,0,0,1,0,0,0,0],
                    [0,0,0,0,0,0,1,0],
                    [0,0,1,0,0,0,0,0],
                    [0,0,0,0,0,0,0,1],
                    [0,1,0,0,0,0,0,0],
                    [0,0,0,0,1,0,0,0],
                    [1,0,0,0,0,0,0,0],
                    [0,0,0,0,0,1,0,0]])
print(matrice)
print(type(matrice))
```

```
[[0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 1]
 [0 1 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0]]
<class 'numpy.ndarray'>
```

Installare pacchetti Python: `pip`

`numpy` non è tra i moduli che abbiamo a disposizione installando Python. Dobbiamo installarlo a parte.

Il modo più comodo per installare un modulo Python esterno è quello di utilizzare `pip`, il *package manager* ufficiale di Python.

Una volta scaricato dal sito https://pypi.org/project/pip/ e installato sul proprio computer, `pip` ci permette di gestire la nostra libreria di pacchetti aggiuntivi per Python.

Ad esempio, possiamo installare `numpy` da linea di comando:

```
pip install numpy
```

Considerazioni

Ogni oggetto di tipo `numpy.ndarray` ha una serie di attributi che lo caratterizzano:

- `size`
- `shape`
- `T` (transpose)
- ...

e di metodi utili:

- `min()`
- `max()`
- `mean()`
- `std()`
- `sort()` (sulle MxN sort delle righe)
- `copy()`
- `where()`
- ...

Possiamo fare oprazioni tra matrici tramite gli operatori `+`, `-`, `*`, `/`, `%`, `**` (se il tipo dati dell'array lo consente), oppure usnado i metodi della classe `numpy.ndarray`.

- `dot()`
- ...

Possiamo fare confronti tra matrici, usando gli operatori di confronto `>`, `<`, `==`, `!=` (se il tipo dati dell'array lo consente). Questo consente anche di creare *maschere* da applicare ai nostri dati.

Numpy permette anche di creare alcune matrici "speciali", usando direttamente i metodi dedicati:

- `arange()`
- `zeros()`
- `ones()`
- `eye()`

Esercizi

Dati i seguenti array:

```
v = np.array([1,2,4,3,5])
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
```

1. Visualizzare gli attibuti `size` e `shape` di `v` e `a`
2. Calcolare massimo, minimo, media e deviazione standard degli elementi di `v`
3. Costruire la matrice trasposta di `a`
4. Oridnare gli elementi di `v`
5. Calcolare il risultato delle operazioni `+`, `-`, `*`, `/` e `**` tra le matrici `a` e `b`
6. Fare il prodotto scalare tra la matrice `a` e la matrice `b`
7. Creare un vettore con la sequenza `0-9`. Creare un vettore con solo i multipli di 3 tra `3` e `27`
8. Creare una matrice di tutti `0`, una matrice di tutti `1` e una matrice diagonale (tutte di 2x2)
9. Trovare ed estrarre solo gli elementi `>2` di `v`
10. Trovare ed estrarre solo gli elementi sulla diagonale di `a`

Esercizi

1. Create una matrice 50x50 piena di 5, con una diagonale di 7
2. Nella matrice:

```
scacchiera = np.array([[0,0,0,1,0,0,0,0],
                       [0,0,0,0,0,0,1,0],
                       [0,0,1,0,0,0,0,0],
                       [0,0,0,0,0,0,0,1],
                       [0,1,0,0,0,0,0,0],
                       [0,0,0,0,1,0,0,0],
                       [1,0,0,0,0,0,0,0],
                       [0,0,0,0,0,1,0,0]])
```

   sostituite tutti gli `1` con `8`.
3. A partire da `scacchiera`, create il vettore:

```
soluzione = np.array([3, 6, 2, 7, 1, 4, 0, 5])
```

`matplotlib`

`matplotlib` è una libreria per creare visualizzazioni (statiche, animate e interattive) in Python. Useremo l'interfaccia **pyplot** che è una collezione di funzioni che rendono `matplotlib` simile a *MATLAB*.

Per questo motivo `pyplot` può essere utilizzato in 2 modi:

- seguendo la struttura degli oggetti che compongono l'immagine
- utilizzando le funzioni di `pyplot` che modificano una figura di *default* memorizzando gli stati successivi che vengono creati dalle chiamate a funzioni che modificano l'immagine attiva

`matplotlib`

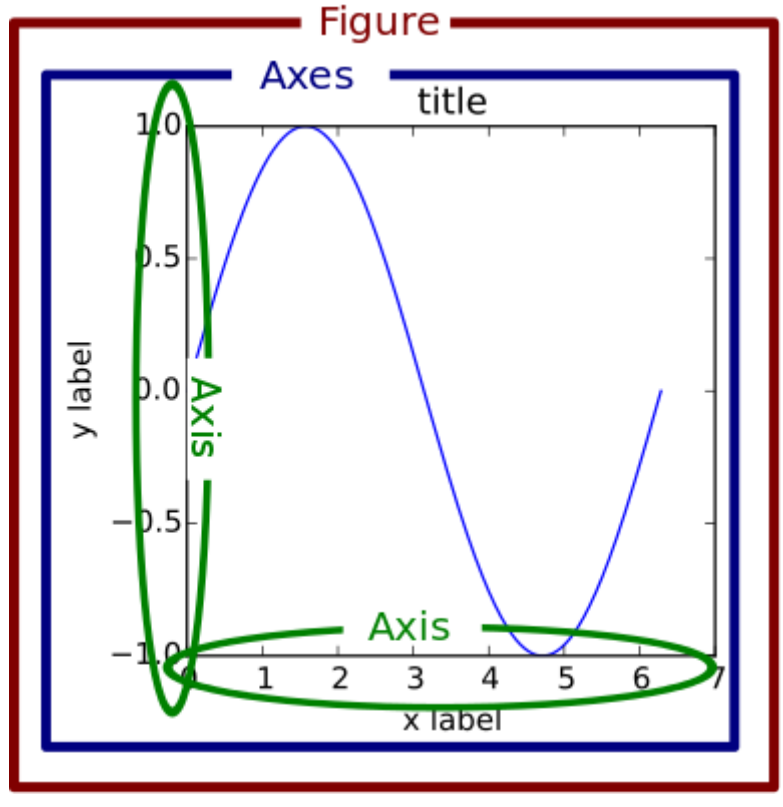
`matplotlib` è una libreria per creare visualizzazioni (statiche, animate e interattive) in Python. Useremo l'interfaccia **pyplot** che è una collezione di funzioni che rendono `matplotlib` simile a *MATLAB*.

Per questo motivo `pyplot` può essere utilizzato in 2 modi:

- seguendo la struttura degli oggetti che compongono l'immagine
- utilizzando le funzioni di `pyplot` che modificano una figura di *default* memorizzando gli stati successivi che vengono creati dalle chiamate a funzioni che modificano l'immagine attiva

> *Anche `matplotlib` è un pacchetto aggiuntivo che va installato a parte:*
>
> ```
> pip install matplotlib
> ```

L'immagine



La struttura dell'immagine generata da `pyplot` è costituita da:

- **Figura**: il contenitore principale
- **Axes**: il grafico (o i grafici) contenuto nella figura
- **Axis**: gli assi specifici di un grafico

Gli oggetti

1) Costruire la *figura*

```
>>> import matplotlib.pyplot as plt
>>> fig, _ = plt.subplots()
>>> type(fig)
<class 'matplotlib.figure.Figure'>
```

> *Nota: in questo caso la seconda variabile ritornata da subplots non ci interessa e la "buttiamo via" con* `_`

2) Prendere il grafico "attivo"

```
>>> graph = fig.gca()  # Get Current Axes
>>> type(graph)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

... oppure tutti i grafici disponibili:

```
>>> graph_list = fig.get_axes()
>>> type(graph_list)
<class 'list'>
```

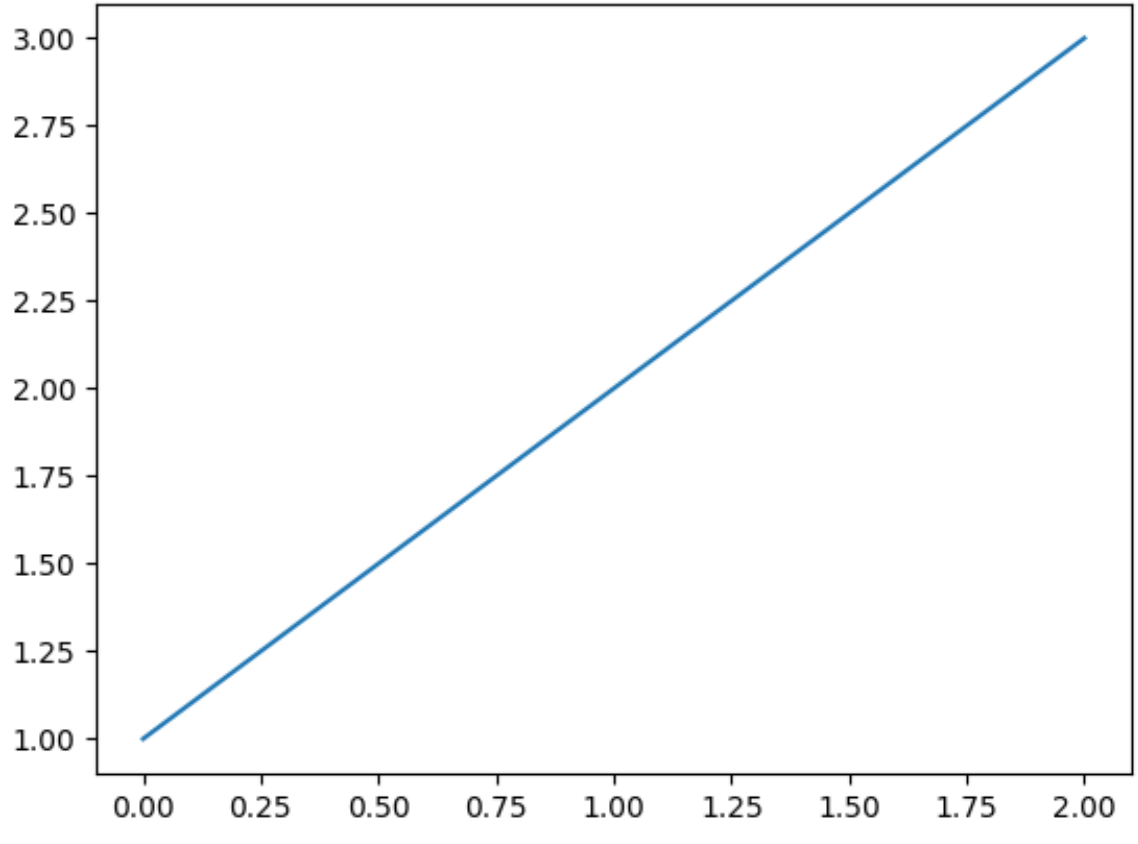
3) Disegnare (plot)

```
>>> graph.plot([1,2,3])
```

4) Salvare o visualizzare l'immagine

```
>>> plt.show()  # matplotlib.pytplot.show per gestire finestra
>>> fig.savefig('name.png')  # per salvare immagine su file
```

`name.png` contiene questo plot:



Più grafici nella figura

Per avere più grafici nella stessa immagine, basta definire in *subplots* la disposizione:

```
>>> fig, _ = plt.subplots(nrows=1, ncols=2)
```

Prendere la lista dei grafici disponibili

```
>>> graph_list = fig.get_axes()
```

Riempirli in ordine (da sinistra)

```
>>> graph_list[0].plot([1,2,3])
>>> graph_list[1].plot([-1,-2,-3])
```

Salvare o visualizzare l'immagine

```
>>> plt.show()  # matplotlib.pytplot.show per gestire finestra
```

Considerazioni: la variabile ”buttata” _

E' una *scorciatoia* per avere gli *axes* (grafici) come array *np*

```
#   fig, _    = plt.subplots(nrows=1, ncols=2)
>>> fig, axes = plt.subplots(nrows=1, ncols=2)
>>> type(axes)
<class 'numpy.ndarray'>

>>> type(axes[0])
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

Ma si può anche usare l'espansione degli argomenti per scrivere:

```
>>> fig, (gr1, gr1) = plt.subplots(nrows=1, ncols=2)
>>> type(gr1)
<class 'matplotlib.axes._subplots.AxesSubplot'>
...

>>> gr1.plot([1,2,3])
>>> gr2.plot([-1,-2,-3])
...
>>> plt.show()  # oppure fig.savefig('nome.png')
```

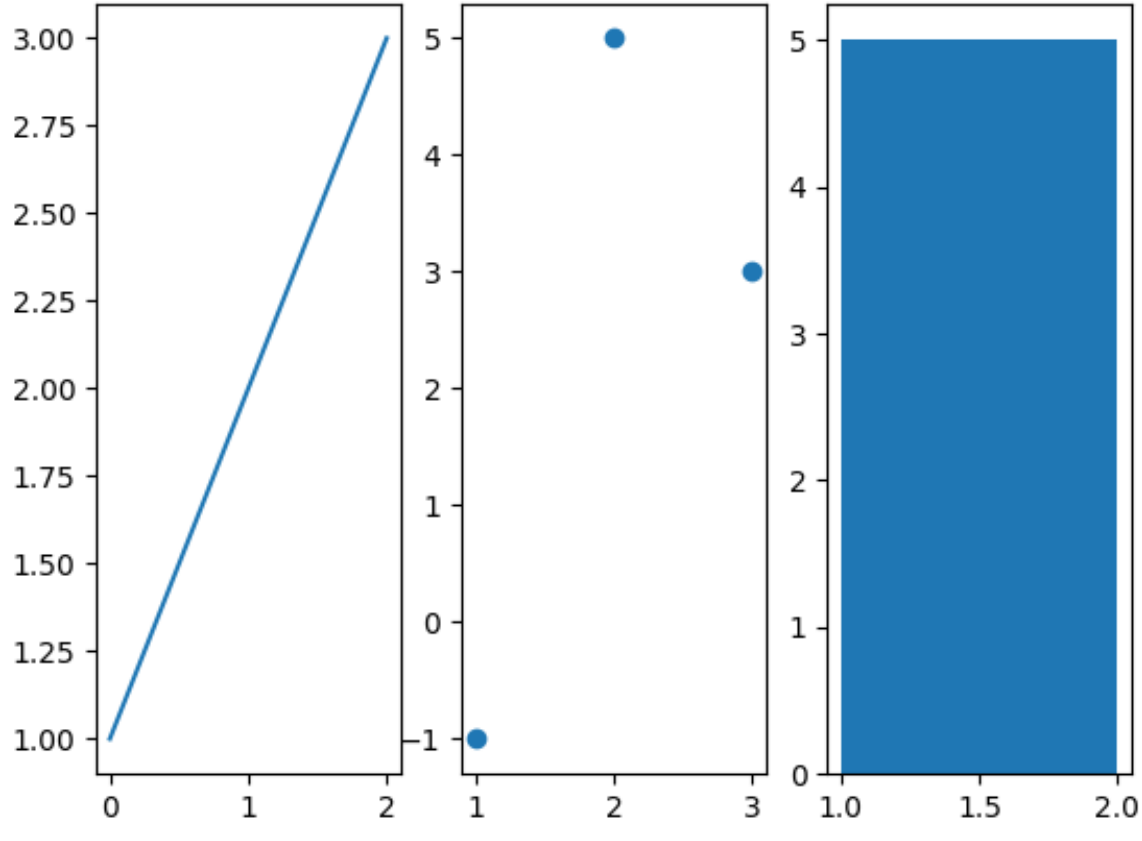
Tipi di grafico

si possono disegnare diversi tipi di grafico con i rispettivi metodi:

```
>>> fig, _ = plt.subplots(nrows=1, ncols=3)
>>> gr_list = fig.get_axes()

>>> gr_list[0].plot([1,2,3])  # line plot
>>> gr_list[1].scatter([1,2,3],[-1,5,3])  # scatter plot (dots)
>>> gr_list[2].hist([1,2,1,2,1,3], np.arange(1,3))  # compute histogram (bars)
...
>>> plt.show()  # oppure fig.savefig('nome.png')
```

Ecco cosa otteniamo:

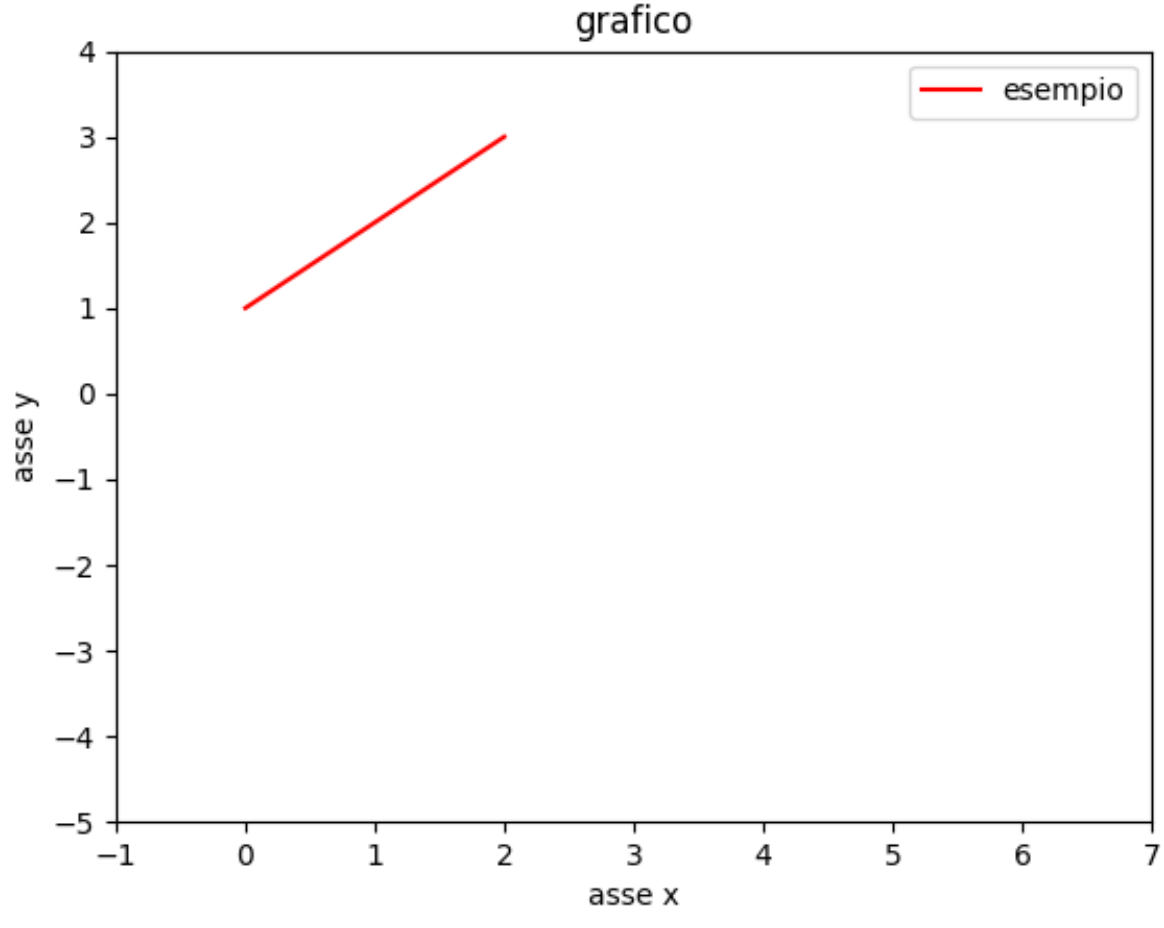


Personalizzare il grafico

Per ogni grafico (axes) si possono modificare gli assi (axis), il titolo, il tipo di tratto, il colore, ecc..

```
>>> fig, _ = plt.subplots(nrows=1, ncols=1)
>>> gr1 = fig.gca()
...
>>> gr1.plot([1,2,3], color='r', label='esempio')
[<matplotlib.lines.Line2D object at 0x7f7ef1a71220>]
>>> gr1.set_xlabel('asse x')   # titolo asse X
Text(0.5, 0, 'asse x')
>>> gr1.set_ylabel('asse y')   # titolo asse Y
Text(0, 0.5, 'asse y')
>>> gr1.set_title('grafico')   # titolo del grafico
Text(0.5, 1.0, 'grafico')
>>> gr1.set_xlim(xmin=-1, xmax=7)  # range plot X
(-1, 7)
>>> gr1.set_ylim(ymin=-5, ymax=4)  # range plotY
(-5, 4)
>>> gr1.legend()     # visualizza legenda con titolo dei datadet
...
plt.show()
```

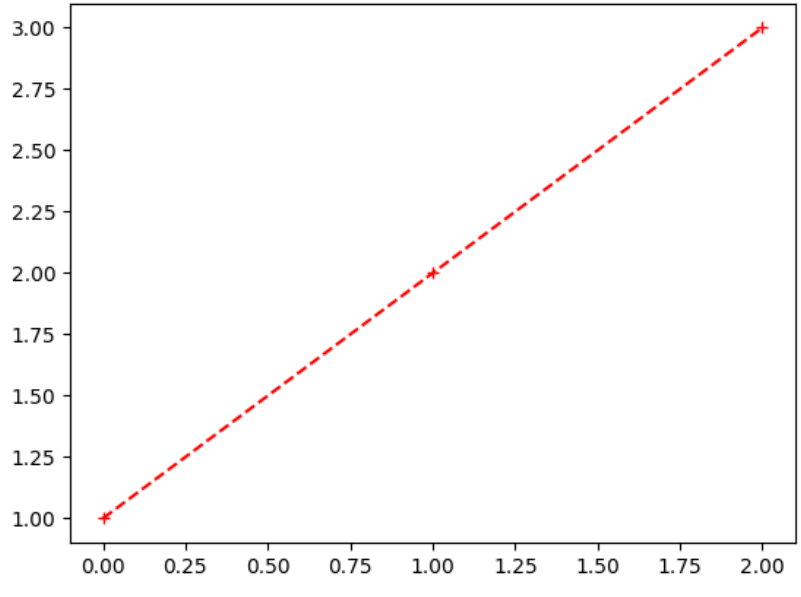
Il nostro plot personalizzato diventa:



Tipo di tratto

La linea con cui è disegnato un grafico è personalizzabile con

```
>>> fig, _ = plt.subplots(nrows=1, ncols=1)
>>> gr1 = fig.gca()
>>> gr1.plot([1,2,3], color='r', linestyle='--',marker='+',label='esempio')
>>> plt.show()
```



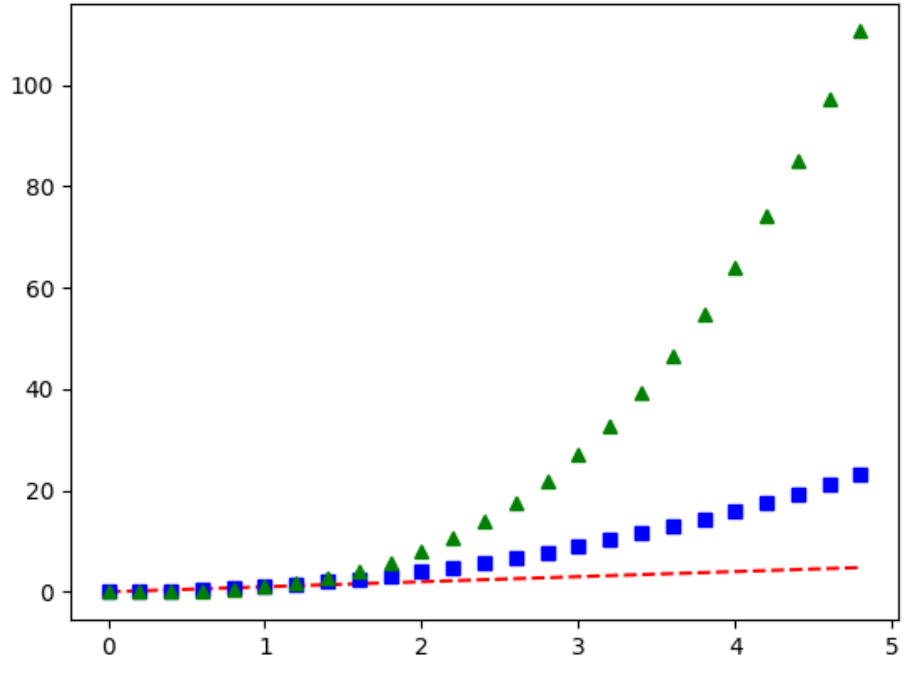
Trovate un elenco di tutte le opzioni nella pagina ufficiale:
https://matplotlib.org/stable/tutorials/introductory/pyplot.html

Considerazioni

Si può disegnare più linee contemporaneamente e usare la notazione MATLAB

```
>>> val = np.arange(0, 5, 0.2)

>>> fig, _ = plt.subplots(nrows=1, ncols=1)
>>> gr1 = fig.gca()
>>> gr1.plot(val, val, 'r--',  val, val**2, 'bs', val, val**3, 'g^')
>>> plt.show()
```



Esercizi

1) Utilizzando `maplotlib`, riprodure il seguente grafico: