Appunti di scuola

Python

Diomede Mazzone

2020, ver. 0.13

Sommario

Moduli

Introduzione

Per affrontare i moduli è necessario ricordare il concetto di procedura e di funzione. Una procedura è uno script a cui è assegnato un nome, definito in modo da essere richiamato ogni volta dovesse servire.

Una funzione è una procedura che evolva a partire da parametri forniti come input ed eventualmente restituisce parametri in output.

La funzione viene richiamata da un programma detto chiamante.

```
# Sintassi generale

# nella definizione bisogna indicare con "def" il nome, e tra parentesi
# gli eventuali parametri di input.
# Se non ci sono parametri di input si devono comunque mettere le
parentesi ().
def nome_funzione (parametro1, parametro2, ….)
   istruzione_1
   istruzione_2
   istruzione_3
   ….
   return risultato  # se non c'è alcun valore di output si omette
l'istruzione return


# Esempio di funzioni

# valore di input: c
# valore di output il risultato di (c * 9 / 5) + 32

def temp(c):
   c=int(input("Enter a temperature in Celsius: "))
```

```
    return (c * 9 / 5) + 32

# programma chiamante
print(temp(c))
```

Tra i vari benefici dell'utilizzo di funzioni possiamo notare:

- Lo sviluppo di funzioni semplifica la leggibilità di un programma, perché riduce gruppi di istruzioni a singole istruzioni da richiamare attraverso un nome ben identificabile.
- Raggruppare un insieme di istruzioni all'interno di una procedura/funzione alleggerisce la scrittura di codice, perchè evita la ripetizione dello stesso gruppo di istruzioni più volte nel programma chiamante.
- L'eventuale correzione di una o più istruzioni interne ad una procedura andrebbe fatta una sola volta, non in tutti i punti del programma chiamante in cui quel blocco sarebbe stato presente se non fosse stato racchiuso in una funzione.

Un programma può includere più funzioni attinenti alle stesse attività e le definizioni possono essere riportate all'interno del medesimo file.

Per utilizzare una stessa funzione in più programmi, invece, è possibile copiarne la definizione e incollarla anche in altri file; questa tecnica è però sconsigliata perché rende laborioso l'aggiornamento dei programmi nei casi in cui occorra una modifica alla funzione stessa: sarebbe necessario andare ad apportare la medesima variazione in tutti quei file nei quali la definizione della funzione è stata incollata. Risulta quindi un buon approccio racchiudere tutte le funzioni utili in un unico file, che sarà chiamato modulo. Questo approccio alla programmazione gode di alcuni vantaggi e prende il nome di **approccio modulare alla programmazione**.

I vantaggi da evidenziare sono:

- **Modularità**: un programma può essere suddiviso in moduli indipendenti, più semplici da gestire;
- **Riutilizzabilità**: lo stesso modulo può essere utilizzato in programmi diversi;

Utilizzo dei moduli

In python (come in molti linguaggi di programmazione) i file contenenti funzioni e altri oggetti utilizzabili da differenti applicazioni sono denominati **moduli**. Per richiamare una funzione definita in un modulo esterno si usa il comando **import**. Ci sono tre diverse modalità di utilizzo del comando:

```
# PRIMO APPROCCIO: import nome_modulo

Import math

print(math.pi)    # stampa il valore pi grego
```

In questo modo verranno inclusi nel programma tutti gli oggetti del modulo importato, quindi si ha accesso a tutte le funzioni ed a tutte le variabili definiti nel modulo. L'accesso a un oggetto si ottiene specificando il modulo di appartenenza come prefisso, secondo la sintassi nome ***modulo.oggetto***, nell'esempio math.pi.

E' possibile includere anche determinati elementi scelti all'interno di un modulo, attraverso la seguente sintassi:

```
# SECONDO APPROCCIO: from nome_modulo import oggetto1, oggetto2

from math import pi

print(pi)
```

In questa modalità si ottiene l'accesso esclusivamente agli oggetti indicati nel comando; per riferirsi a essi all'interno del modulo non c'è bisogno di utilizzare il nome del modulo di origine come prefisso.

Se è necessario includere tutti gli oggetti di un modulo è possibile usare il carattere speciale asterisco **" * "**.

```

```

```
# TERZO APPROCCIO: from nome_modulo import *

from math import *
print(e)
```

In questa modalità si ottiene l'accesso completo a tutti gli oggetti del modulo e non ci sarà bisogno di utilizzare il nome del modulo come prefisso.

Per riferirsi a un oggetto importato con la prima modalità è necessario utilizzare come prefisso il nome del modulo, negli altri due casi l'importazione fa in modo che la variabile e le funzioni importate siano conosciute direttamente nell'ambito del modulo in uso. Bisogna però stare attenti all'eventuale esistenza di elementi con lo stesso nome di quelli importati, perchè potrebbe generare conflitto e quindi mal funzionamenti.

Gestione e salvataggio dei moduli

Il modulo utilizzato tramite il comando import è ricercato dall'interprete all'interno di una lista predefinita di cartelle, dipendente dalla versione di python e dal sistema operativo. È possibile visualizzare la lista delle cartelle attraverso sys.path, una lista di stringhe relative ai path delle cartelle nelle quali sono presenti i moduli.

```
# importare il modulo standard sys ci da accesso al sys.path

import sys
print(sys.path)

# in output avremo


['', '/home/user/Documenti/git/google-chrome', '/usr/bin', '/usr/lib/python38.zip',
'/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload',
'/home/user/.local/lib/python3.8/site-packages', '/usr/lib/python3.8/site-packages']
```

È possibile anche creare dei moduli personali, vanno nominati e salvati in un file .py, questo file andrà salvato in una delle cartelle sopra indicate. Così facendo sarà possibile richiamare questo modulo anche da altre funzioni in esecuzione.

Spazio dei nomi

Per spazio dei nomi intendiamo l'elenco dei nomi e dei corrispondenti valori degli oggetti definiti in un modulo o in una funzione. Definiamo **namespace locale** quello relativo ad un modulo o ad una funzione, il namespace del modulo nel quale la funzione viene chiamata è detto namespace di livello superiore.

In questo modo si viene a creare una gerarchia di namespace uno dentro l'altro, come fossero scatole cinesi. Il namespace più esterno ad ogni applicazione python è quello relativo alle funzioni predefinite, dei moduli e delle parole riservate, ad esempio import o def (parole riservate alla gestione dei moduli, appunto). Questo namespace è chiamato **built-in**.

Le modalità di importazione di un modulo che abbiamo visto, gestiscono diversamente i namespace dei moduli:

- Il primo metodo lascia invariato il namespace locale, per questo è necessario richiamare il nome del modulo prima dell'oggetto.
- Il secondo ed il terzo modo di importazione del modulo fonde i due namespace, questo rende più facile la programmazione perché non bisogna richiamare il modulo. Controindicazione di questo approccio, come è stato notato in precedenza, è che aumenta il rischio di collisione tra i nomi, se gli oggetti del modulo si chiamano allo stesso modo degli oggetti del programma che si sta scrivendo.

Quando si utilizza il primo approccio è possibile utilizzare un alias invece di utilizzare il nome standard del modulo:

```
import math as m
print(m.pi)
```

Visibilità delle variabili

Quando si utilizza una variabile, l'interprete cerca quella variabile prima di tutto nel namespace locale della funzione, all'interno della quale la variabile è stata dichiarata, se non la trova cerca nel namespace che contiene quella funzione e così via fino al namespace built-in.

Una variabile dichiarata in una funzione risulta sconosciuta all'esterno di quella funzione. Bisogna quindi considerare le funzioni come singoli blocchi che permettono una visibilità delle variabili solo all'interno di se stessi, non all'esterno. Questa logica vale per tutti i contenitori, dei moduli fino ai blocchi più piccoli, come le istruzioni racchiuse in un ciclo.

Di conseguenza:

- Se dichiaro una variabile in una funzione, questa non sarà utilizzabile al di fuori di essa.
- Se dichiaro una variabile all'esterno di una funzione, nel programma chiamante, sarà visibile all'interno della funzione. Una variabile dichiarata in questo modo prende il nome di variabile globale.

```
# x variabile globale, dichiarata all'esterno di una funzione
# e visibile anche all'interno della funzione

x = 10
def funzione_esempio():
    y = x
    y += 1
    return y

print(funzione_esempio())

>> 11
```

```
# temp è una variabile locale, dichiarata all'interno di una funzione
# e non visibile all'esterno di essa

def mia_funzione():
    temp = 12
    print(spam)

frutta = temp + 6
>>> NameError: name 'temp' is not defined
```

Variabili dei moduli

assegnare nuovamente un valore. All'interno di un modulo, oltre a dichiarare funzioni, è possibile dichiarare delle variabili con assegnazioni iniziali, così da separare la dichiarazione di queste variabili dal codice.

Un esempio di variabili contenute in un modulo è quello relativo al modulo standard **string**.

La variabile **string.digits** può essere utilizzata per verificare (attraverso un confronto) che tutti i caratteri immessi da tastiera siano di tipo numerico prima di procedere a un'operazione, così da prevenire inserimento di lettere da parte dell'utente.

Analogamente è possibile utilizzare le variabili incluse nel modulo string per verificare si un determinato valore è una lettera minuscola, maiuscola o una rappresentazione esadecimale.

Moduli personali

Per inserire funzioni, variabili o costanti personalizzate, sarà sufficiente inserirle in un file con estensione .py ed importarlo nel programma che dovrà utilizzarle.

In alternativa è possibile salvare il file in una cartella ed includerla nella lista delle cartelle standard da cui i programmi caricano i moduli. La lista delle cartelle vista in precedenza, sys.path, è una normale lista, quindi sarà possibile inserire un elemento come in una lista qualsiasi:

```
sys.path = sys.path + ['c:\python39\moduli_personali']
```

I moduli standard

Python include in modo nativo diverse decine di moduli, per visualizzarli bisogna eseguire il comando **help("modules")**.

Saranno descritti i principali moduli standard di uso comune.

Modulo Math

Il modulo math prevede molte funzioni spesso utilizzate per elaborazioni matematiche.

math.pow(x,y)	Restituisce la potenza con base x disponente y
math.sqrt(x)	restituisce la radice quadrata di x
math.ceil(x)	restituisce il più piccolo intero maggiore o uguale a x
math.floor(x)	restituisce il più grande intero minore o uguale a x
math.factorial(x)	Restituisce il fattoriale di x (con x intero positivo)

L'elenco completo di tutte le funzioni incluse nel modulo Math è possibile visualizzarlo attraverso

help(math) nell'IDLE di Python.

Modulo Random

Spesso durante la programmazione di alcune funzioni può essere utile generare numeri casuali. In realtà i numeri non sono mai puramente casuali ma sempre il frutto di calcoli, seppur complessi e difficilmente prevedibili, per questo motivo vengono detti numeri **pseudo-casuali**. in Python questi numeri sono prelevati da una sequenza predeterminata di $2^{19937}-1$ numeri, questo garantisce con una discreta affidabilità la non ripetizione dei numeri forniti.

La funzione ***random()***, del modulo Random, restituisce quindi un numero pseudo casuale appartenenti all'intervallo che inizia da zero ed arriva a 1 escluso: [0,1).

```
import random
x = random.random()
print(x)

## in output vedremo ##

0.56940303958863
```

Ogni volta che avvieremo lo script nell'esempio precedente genererà un valore diverso. Nella tabella seguente, invece, sono descritte alcune delle funzioni maggiormente utilizzate.

random.randint(a,b)	Restituisce un numero intero appartenente all'intervallo con estremi inclusi, [a,b].
random.choice(oggetto)	Consente di prelevare un elemento a caso tra quelli che compongono l'oggetto indicizzabile passato come parametro.
random.shuffle(lista)	Applicato ad una lista restituisce la stessa lista, ma con gli elementi mescolati a caso.
random.sample(oggetto,n)	Estrae una lista di n oggetti a caso a partire dall'oggetto di m elementi (con n minore o uguale di m) passato come parametro, sia Esso una tupla una lista o insieme.

Modulo Time

Ogni computer considera il tempo a partire da un determinato istante iniziale, vengono così calcolati i secondi complessivi "di vita" del computer. Vuol dire che tutte le date e tempi vengono considerati come un contatore di secondi a partire da quel momento iniziale. La funzione ***time()*** restituisce il numero di secondi a partire dal quel punto iniziale, che prende il nome di ***epoc***. La

funzione ***clock()*** restituisce invece il numero di secondi, ma in formato float, quindi con una precisione inferiore al secondo (anche se non tutti i sistemi lo permettono).

Secondo questa logica per conoscere una data bisogna convertire quel numero di secondi in giorno, mese e anno, questa conversione è realizzata dalla funzione ***gmtime(n)*** del modulo Time. Questa funzione prendi in input un numero intero corrispondente ai secondi trascorsi e restituisce un oggetto di tipo ***struct_time***, equivalente a una tupla di 9 numeri interi aventi il seguente significato:

Indice	Attributo	Significato	Valori ammessi
0	tm_year	Anno	
1	tm_mon	Mese	[1,12]
2	tm_mday	Giorno del mese	[1,31]
3	tm_hour	Ora	[0,23]
4	tm_min	Minuto	[0,59]
5	tm_sec	Secondi	[0,61]
6	tm_wday	Giorno della settimana	[0,6] 0 Domenica, 1 Lunedì...
7	tm_yday	Giorno dell'anno	[1,366]
8	tm_isdst	ora legale	0 Solare, 1 Legale, -1 sconosciuta

esempio di utilizzo del tipo **struct_time**:

```
import time

# Voglio sapere il giorno del mese della data x
time.gmtime(23193723908)[2]

>> 24

# Voglio sapere il giorno del mese della data x
time.gmtime(23193723908)[7]

>> 359
```

Modulo Exception

Prima di eseguire un programma è possibile verificare la presenza di **errori di tipo sintattico** attraverso la voce **check Module** del menu **Run** della IDLE di Python.

Gli errori di tipo **semantico** o **logico**, invece, sono più difficili da prevedere e spesso emergono solo durante l'esecuzione del programma, fornendo risultati inattesi, concludendo l'esecuzione ma in modo inesatto.

Un'altra classe di errori sono quelli detti di **runtime**, molto simili a quelli semantico/logici, ma che restituiscono tentativi di realizzare operazioni non consentite.

Ad esempio non è possibile effettuare la divisione per zero, nel caso dovesse capitare python arresta lo script e restituisce un errore all'utente, probabilmente poco comprensibile.

Nel seguente script si calcola la media dei voti di uno studente:

```
somma = 0
n = 0
domanda = 'inserisci il voto ( zero per fermare): '
voto = input(domanda)

while voto != '0':
      somma = somma + float(voto)
      n = n+1
      voto = input (domanda)
media = somma / n
print( ' la media è: ', media)
```

Il programma restituirà un errore se non viene inserito alcun valore.

```
inserisci il voto ( zero per fermare): 0

Traceback (most recent call last):
    File  "/home/user/Documenti/git/labPy/tutorial/esempi/media.py",
line 12, in <module>
    media = somma / n
ZeroDivisionError: division by zero
```

Attraverso il modulo *Exception*, presente nel sistema, è possibile intercettarli, per gestirli o anche solo per renderli chiari all'utente.

Per intercettare questo tipo di errori bisogna inserirli in una sequenza che si chiama **try-except** come nell'esempio seguente.

```
somma = 0
n = 0
domanda = 'inserisci il voto ( zero per fermare): '

try:
    voto = input(domanda)
    while voto != '0':
          somma = somma + float(voto)
          n = n+1
          voto = input (domanda)
   media = somma / n
   print( ' la media è: ', media)
except:
    print("inserisci almeno un voto")
```

In questo modo sarà possibile intercettare la divisione per zero, così da segnalarla con chiarezza all'utente dello script. Non verrà però distinto alcun errore, restituirà sempre lo stesso messaggio, anche se l'errore dipenderà dall'inserimento di un carattere letterale in luogo di uno numerico previsto.

Nel seguente script, invece, si differenziano i diversi tipi di errore, così da guidare al meglio l'utente.

```
somma = 0
n = 0
domanda = 'inserisci il voto ( zero per fermare): '

try:
    voto = input(domanda)

    while voto != '0':
```

```
            somma = somma + float(voto)
            n = n+1
            voto = input (domanda)

    media = somma / n
    print( ' la media è: ', media)

except ZeroDivisionError:
    print('inserisci almeno un voto')

except ValueError:
    print('non è un numero valido')

except:
   print("errore generico")
```

Si noti che la ricerca di errore generica va sempre messa in coda a tutte le altre.

L'elenco degli errori riconoscibili sono indicate è disponibile nella documentazione on line di python.

In genere, quando si gestiscono gli errori, risulta più utile utilizzare una funzione che restituisce un codice di errore, così da gestirlo internamente al programma, senza utilizzare la funzione print di comunicazione con l'utente, attraverso una stringa di output.

Per rilevare errori logici, quindi con il verificarsi di una condizione, si utilizza la funzione ***raise***, come descritto nel seguente esempio.

```
def verificaVoto(voto):
     try:
          votoSuf= int(voto)
          if (votoSuf < 0) or (votoSuf > 10):
              raise RuntimeError()

          if (votoSuf < 6):
              return 'bocciato'
          elseif:
```

```
            return 'promosso'

    except RuntimeError:
        return 0
    except:
        return -1
```

La funzione così costruita restituirà un codice di errore che potrà eventualmente essere gestito dalla funzione chiamante.

Elaborazione dei dati e grafici

Vettori, matrici ed array di grandi dimensioni sono strumenti fondamentali per l'analisi di dati numerici. Spesso quindi si gestiscono i numeri all'interno di strutture dati in modo che, attraverso l'uso di cicli, è possibile elaborarli e manipolarli in modo da ottenere i risultati voluti.

Quando la mole di dati è grande Invece questo tipo di operazione non è ottimale, perché l'uso di cicli su strutture dati di lunghezza variabile ed eterogenei tra loro, rende le operazioni onerose in termini di tempo e di spazio.

NumPy

Negli ambienti di sviluppo il calcolo scientifico basato su Python vengono utilizzate e strutture dati più efficienti di quelli forniti dal linguaggio base, si utilizza infatti principalmente la libreria NumPy.

Apparentemente le strutture dati sembrano simili a semplici liste, cioè generici contenitori di oggetti. In realtà, invece, gli array appartenenti a NumPy, sono array di tipo omogeneo (tutti i dati di un array sono dello stesso tipo) e di dimensione fissa, quindi non modificabile dopo la creazione. Queste due caratteristiche rendono molto più efficienti le operazioni applicate su strutture di questo tipo.

NumPy, inoltre, Include una vasta collezione di operazioni base e funzioni dedicate alle strutture dati come algoritmi di alto livello relativi all'algebra lineare e a trasformazioni più complesse.

Numpy offre, inoltre, un backend numerico per quasi tutte le librerie scientifiche e tecniche dell'ecosistema Python. Per questi argomenti più specifici è più opportuno consultare il sito ufficiae: http://www.numpy.org.

NumPy Array Object

Come abbiamo detto, la libreria *NumPy* riguarda sostanzialmente le strutture di dati per rappresentare array multidimensionali di dati omogenei. La principale struttura di dati presenti in questa libreria si chiama ***ndarray***, che oltre ai dati conservati nella struttura contiene importanti metadati relativi alla forma, la dimensione, la tipologia ed altri attributi. Per conoscere tutti gli attributi associati ai dati lo si può fare attraverso il comando **help(np.ndarray)** in python oppure **np.array?** nella console *IPython*. Nella seguente tabella sono rappresentati alcuni esempi di attributi.

Attributo	**Descrizione**
Shape	Una tupla che contiene il numero di elementi per ogni dimensione (assi) dell'array.
Size	Il numero totale degli elementi dell'array
Ndim	Il numero delle dimensioni (assi)
nbytes	Il numero di Bytes usati per salvare i dati
dtype	Il tipo di dato degli elementi dell'array

L'esempio che segue, invece, permette di comprendere come accedere agli attributi di un Array.

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

type(data)
>> <class 'numpy.ndarray'>

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

data.ndim
>> 2

data.shape
>> (3, 2)
```

```
data.size
>> 6

data.dtype
>> dtype('int64')

data.nbytes
>> 48
```

Quando si crea un Array numpy, non è più possibile modificare il dtype, è possibile invece crearne una copia con un operazione di casting.

Ci sono diversi modi per generare un Array, dipende dalla sua struttura iniziale e soprattutto dall'uso che se ne deve fare. Per questo motivo il modulo mette a disposizione una grande varietà di metodi utili a generare Array di questo tipo.

Matplotlib

Nel mondo scientifico, come in tanti altri campi, è sempre utile rappresentare i dati sotto forma di grafici, Per avere rappresentazioni sia in 2D che in 3D di grandi quantità di numeri. Per ottenere queste rappresentazioni, è possibile procedere principalmente attraverso due metodologie: elaborare separatamente i grafici dopo aver acquisito i dati, oppure generare grafici in modo automatico a partire dai dati. In Python è possibile elaborare in modo automatico i grafici e la più popolare e generica libreria di generazione dei grafici è **Matplotlib** (www.matplotlib.org).

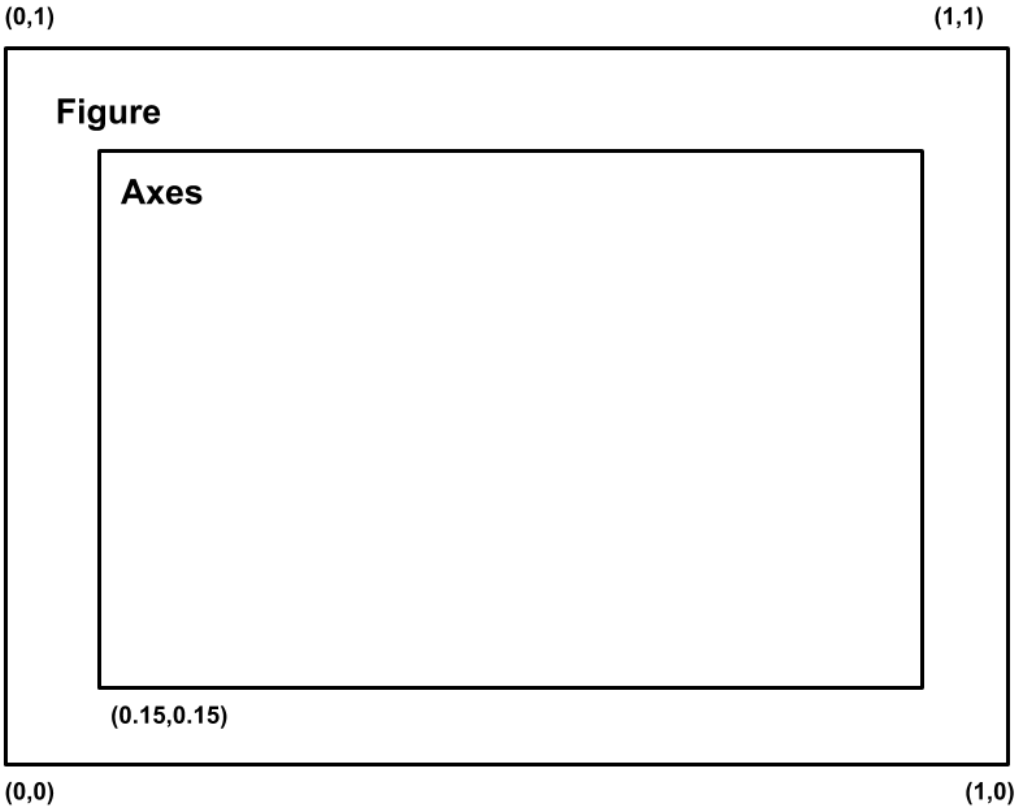
Esistono diversi modi per utilizzare *Matplotlib*, negli esempi che seguiranno verrà importata esattamente come gli altri moduli, così da poterla utilizzare in *script* generici interagendo anche con altri moduli e librerie.

Bisogna Inoltre sottolineare che questa libreria non contiene i soltanto funzioni per generare grafici, ma continui anche supporti per la visualizzazione i grafici in differenti ambienti, quindi la loro esportazione in diversi formati di file come png, pdf, swg.

Quando bisogna richiamare gli strumenti software di backend, per creare ad esempio una finestra in cui rappresentare il grafico, è necessario richiamare la funzione **plt.show()**, così da creare una finestra che alla chiusura terminerà lo script che l'ha generata (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.show.html).

Struttura dei grafici

Un grafico, in *Matplotlib*, è strutturato come un'istanza chiamata **Figure**, ed uno o più Istanze di **Axes** all'interno della figura. Il concetto di istanza deriva dal fatto che la figura e gli assi sono *oggetti* che racchiudono in sé caratteristiche e metodi, detti anche funzioni, che permettono la loro manipolazione. *Figure* racchiude un'area detta **Canvas** che rappresenta lo spazio di lavoro, le istanze *Axes* forniscono un sistema di coordinate (utili ai grafici) che sono assegnate a regioni fisse dell'intera area di lavoro.

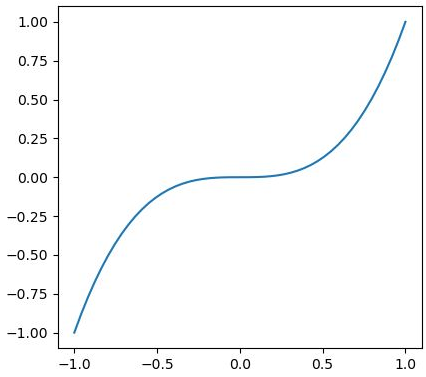


Nella figura precedente sono rappresentate le due istanze, *Figure* determina un'area di lavoro e *Axes* determina lo spazio all'interno del quale verranno collocati gli assi cartesiani. Il sistema di coordinate che prevede la coppia (0,0) in basso a sinistra e la coppia (1,1) in alto a destra permette la collocazione degli *Axes* e quindi dei grafici. Queste coordinate vengono utilizzate soltanto quando si colloca un elemento nell'area di disegno.

Se sono presenti più grafici (quindi più istanze *Axes*) all'interno della stessa area di disegno, possono essere collocati in modo arbitrario all'interno dell'area attraverso le coordinate, oppure si può procedere in modo automatico utilizzando gli strumenti messi a disposizione del modulo.

Si osservi il seguente esempio molto semplice.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-1.0,1.0,50,endpoint=True)
y = x**3
plt.plot(x,y)
plt.show()
```



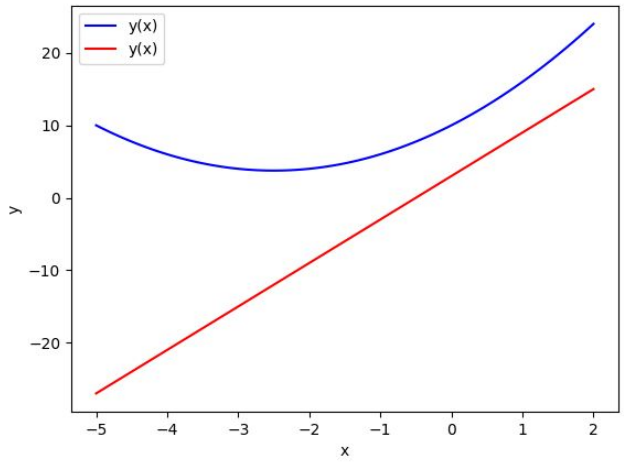
Si osservi che:

- La dimensione degli assi combacia.
- I **tick mark** (si chiamano così i punti evidenziati degli assi) degli assi sono tutti distanziati di 0.5 unità.
- Non c'è titolo e non ci sono etichette sugli assi.
- Non c'è una leggenda.
- Il colore della linea del grafico è blu.

Queste sono le impostazioni di default, si vedrà in seguito come personalizzarle. Si osservi invece l'esempio che segue, più articolato: due equazioni inserite nello stesso grafico.

```
## questo esempio è presente in http://github.com/doceo/labPy/matplotlib
# grafico-funzioni-1.py

import matplotlib.pyplot as plt
```

```
import numpy as np

# https://numpy.org/doc/stable/reference/generated/numpy.linspace.html
x = np.linspace(-5,2,100)

y1 = x**2 + 5*x +10
y2 = 6*x + 3

fig, ax = plt.subplots()
ax.plot(x,y1,color="blue", label="y(x)")
ax.plot(x,y2,color="red", label="y(x)")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.legend()
plt.show()
```



Le variabili *y1* e *y2*, si noti, sono espressioni della variabili *x*, che a sua volta è un oggetto di tipo *numPy*. La variabile *x* è un array di tipo *np* generato dal metodo **linspace()**, che restituisce numeri equidistanti su un intervallo specificato (nel caso specifico sono 100 punti compresi tra -5 e 2). Successivamente viene usata la funzione **plt.subplot()** per generare l'istanza di Figure e di Axes. Questa funzione quindi risulta conveniente perché attraverso di essa possiamo generare contemporaneamente l'area di lavoro e le istanze dei grafici. A questo punto sarà necessario utilizzare i metodi relativi ad *Axes*, così da creare i grafici delle equazioni scritte nelle prime righe di codice. Per fare ciò è stato utilizzato il metodo **ax.plot()**, che prende come primo e secondo argomento array di tipo *numPy*, quindi i dati relativi ai valori X e Y dei punti dei grafici, *subplot* unisce tutti questi punti in forma di linea all'interno del grafico. Inoltre è stato usato il parametro **color** come argomento opzionale per specificare il colore della linea e il parametro **label** per indicare i valori nella legenda. Per indicare invece la legenda relativa agli assi, quindi per

distinguere l'asse delle x dall'asse delle y, Sono stati introdotti altri due metodi: **set_xlable("x")** e **set_ylable("y")**. Come è possibile comprendere dal codice, sia i parametri all'interno della funzione *plot* che i parametri di input per *set_xlable("x")* sono di tipo stringa. A conclusione dello script, *plt.show()* genererà la finestra con i grafici richiesti.

È possibile immaginare, quindi, che esista un'ampia gamma di parametri che permettono una completa personalizzazione dei grafici realizzabili.

Figure

Come abbiamo detto l'oggetto *Figure* è necessario per rappresentare un grafico, perchè genera uno spazio di lavoro destinato a contenere gli oggetti di tipo *Axis*. *Figure* possiede molte proprietà e metodi per ottimizzare questo processo, in particolare è possibile fornire le dimensioni precise dello spazio di lavoro attraverso l'attributo **figsize**, assegnandogli tuple intese come coppia (width, height), l'unità di misura è il pollice (non il centimetro). Spesso, Inoltre, si determina il colore dello spazio di lavoro attraverso l'attributo **facecolor**.

Una volta creato lo spazio di lavoro bisogna usare il metodo **add_axes** per creare una nuova istanza *Axis* e da collocare in una posizione all'interno dello spazio di lavoro. *add_axes* prende in input come argomenti una lista contenente le coordinate che, partendo dell'angolo in basso a sinistra per arrivare a quello in alto a destra, determinano la posizione dell'oggetto Axes.

Ad esempio, se si assegna la lista (0, 0, 1, 1) all'oggetto Axis, verrebbe occupata l'intera area di lavoro, senza lasciare spazio alla legenda o ad altri elementi utili. Per tale motivo è buona norma utilizzare una lista diversa, ad esempio (0.1, 0.1, 0.8, 0.8). In questo modo l'oggetto occuperebbe il centro dello spazio di lavoro corrispondente a una copertura del 80%.

Dopo aver creato le istanze *Figure* e *Axes* e collocato *Axes* dentro la *Figure*, attraverso *ass_axes*, è possibile rappresentare i dati utilizzando i metodi messi a disposizione dall'oggetto *Axes*. Prima di esplorare le possibilità relative all'oggetto *Axes*, è bene sottolineare le tante possibilità che figura mette a disposizione per la creazione, la manipolazione e la stampa di grafici, una volta personalizzati attraverso i metodi di *Axes*. Ad esempio:

Metodo	Descrizione	Esempio
suptitle()	Inserisce un titolo al centro dell'area di lavoro, Il titolo in sito come stringa nel primo parametro di input	`fig.suptitle('This is the figure title', fontsize=12)`

savefig()	Salva il grafico in un file, il nome del file deve essere passato come primo parametro (*fname* nell'esempio). Esistono molti parametri da poter passare a questo metodo, anche se non sono obbligatori. L'estensione del file, di default, e determinata dal formato indicato nella stringa passato come primo parametro, è possibile comunque personalizzare l'estensione del file attraverso il parametro *format*. (png, pdf, eps ed svg sono tutte estensioni accettate. L'argomento dpi, invece, definisce la risoluzione. Nunzia ti viene sotto definisco la comédie su Subito plt Shih Tzu nice to be happy with The quality of life And finally the figures che non serviva l'apparecchio si trasforma in Show ARP table fixed column for the listing Please tecniche del salto in su	`savefig(fname, dpi=None, facecolor='w', edgecolor='w', orientation='portrait', papertype=None, format=None, transparent=False, bbox_inches=None, pad_inches=0.1, frameon=None, metadata=None)`

Tutti i metodi messi a disposizione dall'oggetto *Figure*, sono ben descritti nella documentazione di riferimento:

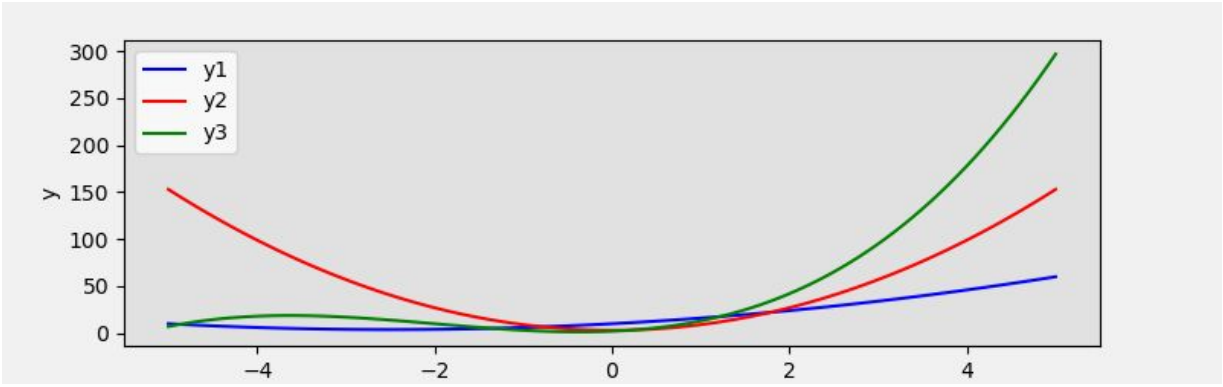
https://matplotlib.org/3.1.1/api/figure_api.html?highlight=figure#module-matplotlib.figure

Un esempio di utilizzo di parametri è disponibile attraverso il seguente codice:

figure-esempio.py

```
## esempio riportato sugli appunti condivisi

import matplotlib.pyplot as plt
import numpy as np
```

```
# i colori vanno passati come stringa,
# il colore è ricavato dalla codifica esadecimale RGB
fig = plt.figure(figsize=(8,2.5), facecolor="#f5f5f5")


# la posizione di Axes la determiniamo come distanza dai bordi
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
ax = fig.add_axes((left, bottom, width, height), facecolor="#e1e1e1")

x = np.linspace(-5,5,100)

y1 = x**2 + 5*x +10
y2 = 6*x**2 + 3
y3 = x**3 + 6*x**2 + 4*x+2

ax.plot(x,y1,color="blue", label="y1")
ax.plot(x,y2,color="red", label="y2")
ax.plot(x,y3,color="green", label="y3")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.legend()

# salva il grafico in un file di nome grafico.png
fig.savefig("grafico.png", dpi=100, facecolor="#f1f1f1")
plt.show()
```



Axes

Le istanze degli oggetti *Axes*, come è stato introdotto precedentemente, sono collocati all'interno delle istanze di oggetti *Figure*. Gli *Axes*, dunque, sono il cuore del modulo *Mathplotlib*, perché attraverso di essi è possibile manipolare i grafici che rappresentano l'elaborazione numerica e, sempre attraverso questi oggetti, è possibile scegliere il tipo di grafico per la rappresentazione da realizzare.

È stato utilizzato più volte, negli esempi precedenti, il metodo *add_axes*, sottolineando come questo metodo fosse particolarmente duttile nel posizionare sia manualmente che automaticamente i grafici all'interno dell'area di lavoro. Questo tipo di necessità emerge soprattutto quando bisogna rappresentare più grafici in una griglia, all'interno di una sola istanza *Figure*. Esistono diverse modalità per gestire rappresentazioni complesse, ma per motivi di semplicità verrà descritta solo quella relativa alla funzione **plt.subplots()**.

Per impostare una griglia di grafici bisogna passare a questa funzione due parametri: **nrows** e **ncolos**. Questi due valori permetteranno a *subplot* di generare una griglia di *nrows* righe per *ncolos* colonne.

```
git, axes = plt.subplots(nrows=3, ncols = 3)
```

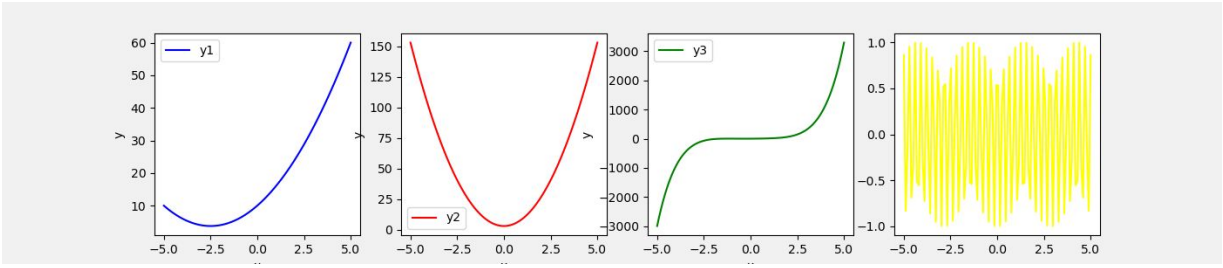
La funzione subplot restituirà una tupla **(fig, axes)**, dove *fig* è una istanza *Figure* e *axes* è un array *NumPy* di dimensione (nrows, ncols, che rappresentano righe e colonne), i cui elementi sono istanze di tipo *Axes*, da collocare nell'area di lavoro.

È possibile concludere quindi che il metodo *subplots* permette di gestire più grafici nella stessa area di lavoro restituendo le istanze necessarie sia relative a *Figure* che ad *Axes*. Si noti infatti, nell'esempio che segue, che i grafici vengono inseriti nella matrice chiamata *ax*, così da permettere una gestione più semplice di dati comuni a tutti i grafici. Nell'esempio, infatti, viene applicato un ciclo di *for* per indicare a tutti i grafici le etichette relative agli assi cartesiani.

subplot-griglia.py

```
import matplotlib.pyplot as plt
import numpy as np
```

```
# indichiamo una matrice di una riga per 4 colonne
fig, ax = plt.subplots(1, 4, figsize=(14,3))

# la posizione di Axes la determiniamo come distanza dai bordi
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
x = np.linspace(-5,5,100)

y1 = x**2 + 5*x +10
y2 = 6*x**2 + 3
y3 = x**5 + 6*x**2 + 4*x+2
y4 = np.cos(20*x)

ax[0].plot(x,y1,color="blue", label="y1")
ax[1].plot(x,y2,color="red", label="y2")
ax[2].plot(x,y3,color="green", label="y3")
ax[3].plot(x,y4,color="yellow", label="y4")

for i in range(3):
   ax[i].set_xlabel("x")
   ax[i].set_ylabel("y")
   ax[i].legend()

# salva il grafico in un file di nome grafico.png
fig.savefig("grafico.png", dpi=100, facecolor="#f1f1f1")
plt.show()
```



Osservando l'esempio precedente emerge l'utilizzo del metodo **plot()** per assegnare all'oggetto *Axis*, appartenenti alla matrice *ax*, il grafico da rappresentare. La forma del diagramma è in

funzione degli input che riceve, se ad esempio diamo un solo parametro ingresso lui costruirà il grafico in cui sull'asse Y ci saranno gli elementi del vettore dato in ingresso e sull'asse X ci saranno gli indici di tale vettore. Se invece diamo in ingresso a *plot* due vettori, lui inserirà sull'asse X il primo vettore è sull'asse Y il secondo, così da generare i punti del diagramma e di conseguenza la sua linea.

Personalizzare i grafici

Il metodo plot, mette a disposizione molti parametri, per gestire i grafici e personalizzare la visualizzazione, nella tabella che segue sono elencati solo alcuni di questi.

Argomento	Esempio	Descrizione
color	va determinato attraverso una stringa identificativa del colore: “red”, “blue”, oppure un colore RGB in formato esadecimale (#f3f3f3)	colore della linea
alpha	indica la trasparenza con un numero reale compreso tra 0.0 (totalmente trasparente) ed 1.0 (totalmente opaco)	trasparenza
linewidth, lw	numero reale	spessore della linea
linestyle, ls	“-” linea continua “--” linea tratteggiata “:” linea punteggiata “.-” alternanza di tratto e punto	lo stile della linea del grafico
marker	+,o,* = croce, cerchi e stelle s = quadrato . = punto piccolo 1,2,3,4..= forme triangolari	ogni punto può essere rappresentato con un simbolo
markersize	numero reale	la grandezza del punto (marker)
markerfacecolor	valori identificativi di colori	il colore del punto (marker)
markeredgewidth	numero reale	larghezza della linea che definisce il marker

markeredgecolor	valori identificativi dei colori	colore della linea di definizione del marler

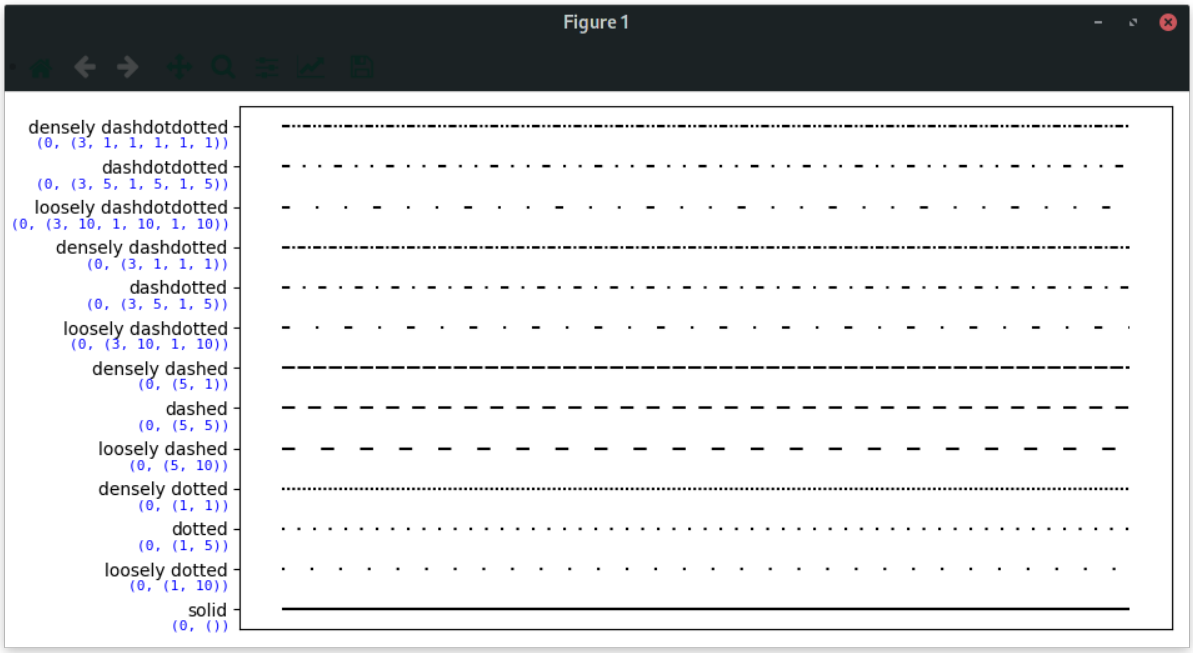
Il grafico di default prevede una semplice linea, è chiaro però che se il numero di funzioni aumenta, al fine di rendere leggibile un grafico è necessario cambiare non soltanto il colore, ma anche il tipo di linea che può diventare tratteggiata, una sequenza di punti, una linea spezzata oppure cambiare proprio la natura del grafico, rappresentandolo come grafico a barre, a dispersione oppure come aree di superficie colorata.

Attraverso il seguente script è possibile vedere tutte le possibili personalizzazioni grafiche relative alle linee.

linestyle.py

```
"""
==========
Linestyles
==========

https://matplotlib.org/examples/lines_bars_and_markers/linestyles.html

This examples showcases different linestyles copying those of Tikz/PGF.
"""
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict
from matplotlib.transforms import blended_transform_factory

linestyles = OrderedDict(
    [('solid',               (0, ())),
     ('loosely dotted',      (0, (1, 10))),
     ('dotted',              (0, (1, 5))),
     ('densely dotted',      (0, (1, 1))),

     ('loosely dashed',      (0, (5, 10))),
```

```
    ('dashed',                (0, (5, 5))),
    ('densely dashed',        (0, (5, 1))),

    ('loosely dashdotted',    (0, (3, 10, 1, 10))),
    ('dashdotted',            (0, (3, 5, 1, 5))),
    ('densely dashdotted',    (0, (3, 1, 1, 1))),

    ('loosely dashdotdotted', (0, (3, 10, 1, 10, 1, 10))),
    ('dashdotdotted',           (0, (3, 5, 1, 5, 1, 5))),
    ('densely dashdotdotted', (0, (3, 1, 1, 1, 1, 1)))])


plt.figure(figsize=(10, 6))
ax = plt.subplot(1, 1, 1)

X, Y = np.linspace(0, 100, 10), np.zeros(10)
for i, (name, linestyle) in enumerate(linestyles.items()):
   ax.plot(X, Y+i, linestyle=linestyle, linewidth=1.5, color='black')

ax.set_ylim(-0.5, len(linestyles)-0.5)
plt.yticks(np.arange(len(linestyles)), linestyles.keys())
plt.xticks([])

# For each line style, add a text annotation with a small offset from
# the reference point (0 in Axes coords, y tick value in Data coords).
reference_transform = blended_transform_factory(ax.transAxes,
ax.transData)
for i, (name, linestyle) in enumerate(linestyles.items()):
   ax.annotate(str(linestyle), xy=(0.0, i),
xycoords=reference_transform,
                xytext=(-6, -12), textcoords='offset points',
color="blue",
                fontsize=8, ha="right", family="monospace")

plt.tight_layout()
plt.show()
```



Volendo sintetizzare il ruolo del modulo *pyplot*, potremmo considerarlo una raccolta di funzioni che consentono di utilizzare le funzionalità di matplotlib. Ogni funzione di *pyplot* agisce in una singola finestra di plot, detta Figura, ad esempio:

- crea una figura
- crea un'area di plotting all'interno di una figura
- disegna dei grafici nell'area di plotting
- personalizza il plot con etichette e altri elementi grafici

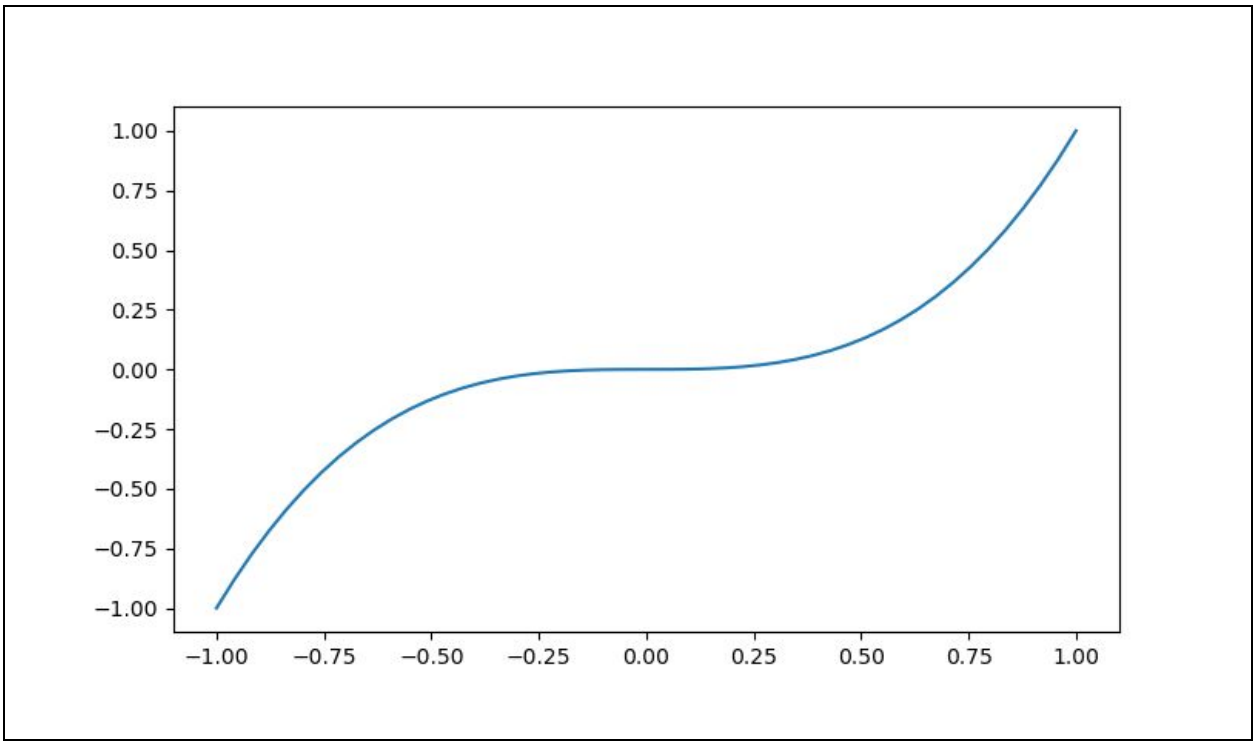
Bisogna sottolineare, inoltre, che **pyplot è stateful**, ovvero tiene traccia dello stato della figura corrente e della relativa area di disegno, le sue funzioni agiscono sulla figura corrente.

Personalizzazione degli assi

La funzione **pyplot.xticks()** (o **.yticks()**) permette di modificare il comportamento di *pyplot* relativo ai *tick mark* dell'asse X e Y.

Applicando questi valori di ingresso alla funzione si otterrebbe il grafico in figura, con i *tick mark* distanziati di 0.25.

```
plt.xticks([0.25*k for k in range(-4,5)])
plt.yticks([0.25*k for k in range(-4,5)])
```



L'argomento da passare a questa funzione, comunque può anche non essere una lista equispaziata, ma una lista di punti scelti in relazione dei dati da rappresentare. Sarà possibile, inoltre, passare anche una etichetta da associare ai singoli tick del relativo asse.

Legenda

Un grafico con più linee necessita sicuramente di una leggenda chi spieghi con un etichetta le diverse curve rappresentate. È possibile dotare ogni istanza *Axes* di una legenda attraverso il metodo **legend**, così da includere la descrizione di ogni curva che viene inclusa a partire dall'argomento *lable* passato alla funzione *Axes.plot*. il metodo *legend* permette di agire su moltissimi parametri, per i dettagli si consiglia di accedere all'help (help(plt.legend)). Di default gli argomenti di una legenda sono disposti in un'unica colonna, ma è possibile personalizzare la descrizione in più colonne. Un'altra personalizzazione molto spesso necessaria è la posizione della leggenda stessa, è possibile, infatti, attraverso il parametro **loc** posizionarla nei quattro angoli; *loc=1* indica l'angolo in alto a destra, *loc = 2* indica l'angolo in alto a sinistra e così via seguendo un ordine antiorario.