Calcolo scientifico con Python (autore: Vittorio Albertoni)

Premessa

Questo manualetto costituisce prosecuzione ideale di quello, nominato «python», allegato al mio articolo «Python per principianti» dell'ottobre 2020, archiviato in PROGRAMMAZIONE sul mio blog *www.vittal.it*, dove ho illustrato le basi del linguaggio di programmazione Python riferendomi alla dotazione di base di strumenti disponibili alla prima installazione dell'interprete sul nostro computer.

In questa dotazione di base troviamo il modulo **math** che contiene molte funzioni matematiche preconfezionate adatte per calcoli scientifici e ne ho parlato nel citato manualetto.

Sempre la dotazione di base ci offre strutture di dati complesse assimilabili a ciò che in informatica va sotto il nome di array (`liste`, `tuple`) e, pur con parecchio sforzo, già utilizzando queste strutture potremmo sviluppare calcoli vettoriali e matriciali.

E' proprio per semplificare e rendere più efficienti queste tipologie di calcolo che è stato creato il modulo **numpy**, che è diventato il fulcro di una serie di altri moduli per il calcolo scientifico (**scipy** e **matplotlib**) e per il così detto machine learning (**pandas** e **sklearn**).

Con il machine learning entriamo in un'area poco abbordabile da parte di principianti e dilettanti e di un certo impegno per gli stessi professionisti addetti ai lavori. Fortunatamente, sempre nel mondo Python, esiste lo strumento **Orange** che semplifica la vita a tutti: l'ho illustrato in allegato al mio articolo «Orange: data science con Python senza scrivere codice» del dicembre 2020, archiviato nel mio blog *www.vittal.it* nella categoria SOFTWARE LIBERO.

Per il calcolo scientifico, invece, anche principianti e dilettanti si possono divertire almeno attraverso un utilizzo delle cose più semplici che troviamo nei moduli numpy, scipy e matplotlib e questo manualetto ci fa vedere come.

Per chi fosse interessato, segnalo qui i miei altri interventi in tema di Python sul blog all'indirizzo *www.vittal.it*:
. «Python su Android» del giugno 2015, con allegato il manualetto PDF «sl4a», archiviato in PROGRAMMAZIONE,
. «Python per tutti» del febbraio 2017, con allegato il manualetto PDF «mondo_python», archiviato in PROGRAMMAZIONE,
. «Grafica con Python» del maggio 2018, con allegato il manualetto PDF «tkinter», archiviato in PROGRAMMAZIONE,
. «Ancora grafica con Python» dell'ottobre 2018, con allegato il manualetto PDF «grafica_interattiva_python», archiviato in PROGRAMMAZIONE,
. «Software libero per data scientists» dell'aprile 2019, con allegato il manualetto PDF «python_anaconda», archiviato in SOFTWARE LIBERO,
. «Ancora Python su Android» del maggio 2019, con allegato il manualetto PDF «pydroid», archiviato in PROGRAMMAZIONE,
. «Software libero per il calcolo simbolico» del settembre 2019, con allegato il manualetto PDF «simpy», archiviato in PROGRAMMAZIONE,
. «La blockchain secondo Python» del novembre 2019, con allegato il manualetto PDF «blockchain_python», archiviato in PROGRAMMAZIONE.

Indice

1 NumPy **3**
- 1.1 ndarray . . . 3
  - 1.1.1 Proprietà . . . 5
  - 1.1.2 Metodi . . . 6
- 1.2 Operazioni matematiche . . . 7
- 1.3 Calcolo matriciale . . . 8

2 Matplotlib **10**

3 SciPy **11**
- 3.1 Costanti . . . 11
- 3.2 Algebra lineare . . . 12
- 3.3 Interpolazione . . . 12
- 3.4 Integrazione . . . 14
- 3.5 Ottimizzazione . . . 16
- 3.6 Statistica . . . 17

1 NumPy

NumPy sta per Numeric Python e, prendendo la definizione di Wikipedia, è una libreria open source per il linguaggio di programmazione Python, che aggiunge supporto a grandi matrici e array multidimensionali insieme a una vasta collezione di funzioni matematiche di alto livello per poter operare efficientemente su queste strutture dati.

All'indirizzo *https://numpy.org/* troviamo tutto ciò che riguarda NumPy e il suo utilizzo: documentazione, tutorial, ecc.

Prerequisito per l'utilizzo di NumPy è avere installato Python.

Prima di installare NumPy possiamo verificare se non sia per caso già installato digitando nella shell di Python `import numpy`: se non esce alcun segnale di errore significa che NumPy è già installato.

L'installazione può facilmente avvenire con pip[1].

Per usare NumPy dobbiamo importarlo all'inizio dello script e, dei tre modi per poterlo fare, il più efficiente è

```
import numpy as np
```

che richiede poi il richiamo delle funzioni attraverso la sintassi
`np.`<nome_funzione>[2].

1.1 ndarray

Sta per n-dimension-array ed è la struttura dati di base per NumPy. E' un oggetto che rappresenta un array multidimensionale e omogeneo di dimensioni fisse.

Se l'array è ad una dimensione si tratta di un vettore; se è a due dimensioni si tratta di una matrice e teoricamente non c'è limite alle dimensioni, anche se sul piano pratico si va difficilmente oltre queste due.

Avendo importato il modulo NumPy come indicato prima, l'array si crea con la funzione

```
np.array()
```

il cui argomento, da mettere tra le parentesi tonde, può essere una tupla o una lista.

Costruiamo il vettore

```
1 2 3
```

con

```
np.array((1,2,3))
```

oppure

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

Costruiamo la matrice

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

con

```
np.array(((1,2,3), (4,5,6), (7,8,9)))
```

oppure

```
np.array([[1,2,3], [4,5,6], [7,8,9]]).
```

Come si vede, il vettore è rappresentato da una sola tupla o da una sola lista, mentre la matrice è rappresentata da una tupla di tuple o da una lista di liste i cui elementi corrispondono alle righe della matrice.

---

[1] In proposito si veda l'allegato PDF «mondo_python» al mio articolo «Python per tutti» del febbraio 2017, archiviato in PROGRAMMAZIONE sul mio blog all'indirizzo *www.vittal.it*.

[2] L'importazione con i comandi `import numpy` o `from numpy import *`, pur possibile, è sconsigliata in quanto può dare origine a confusione sui nomi delle funzioni e, almeno nel primo caso, allunga anche i tempi di riferimento.

La funzione array accetta un ulteriore argomento facoltativo con il quale possiamo stabilire il tipo di dato numerico (`int` o `float`). Se esso non viene indicato, automaticamente viene desunto da quanto inserito: se un solo dato è di tipo float, tutti saranno di tipo float.

Esistono particolari funzioni per creare array di un certo tipo.

`np.arange(`<partenza>, <arrivo>, <step>`)`
crea un vettore che contiene i numeri compresi tra partenza e arrivo - 1 cadenzati secondo lo step. Se si indica solo l'arrivo si crea un vettore che contiene i numeri da 0 a arrivo - 1. Se non si indica lo step i numeri saranno indicati per unità consecutive.
`np.arange(5)` crea il vettore `0 1 2 3 4`
`np.arange(1, 5)` crea il vettore `1 2 3 4`
`np.arange(1, 3, 0.5)` crea il vettore `1 1,5 2 2,5`
`np.arange(1,11,2)` crea il vettore `1 3 5 7 9`

`np.linspace(`<partenza>, <arrivo>, <quantità_punti>`)`
crea un vettore che suddivide l'enumerazione tra partenza e arrivo in tanti punti quanti indicati.
np.linspace(1,5,5) crea il vettore 1 2 3 4 5
np.linspace(1,5,3) crea il vettore 1 3 5

`np.zeros(`<elementi>`)`
crea un vettore con il numero di elementi indicato come argomento, tutti uguali a 0; se indichiamo gli elementi con una tupla contenente il numero di righe e di colonne di una matrice creiamo una matrice di zeri.
`np.zeros(4)` crea il vettore `0 0 0 0`
`np.zeros((2,3))` crea la matrice

$$\begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

`np.ones(`<elementi>`)`
crea un vettore con il numero di elementi indicato come argomento, tutti uguali a 1; se indichiamo gli elementi con una tupla contenente il numero di righe e di colonne di una matrice creiamo una matrice di 1.
`np.ones(4)` crea il vettore `1 1 1 1`
`np.ones((2,3))` crea la matrice

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix}$$

Anche queste funzioni accettano il parametro opzionale per stabilire il tipo di dato (`int` o `float`). Se esso non viene indicato viene attribuito il tipo float.

L'oggetto ndarray può essere inserito in una variabile.
`a = np.array([1,2,3])`
crea la variabile a contenente il vettore `1 2 3`,
`m = np.array([[1,2,3,4], [5,6,7,8]])`
crea la variabile `m` contenente la matrice

$$\begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{vmatrix}$$

Ogni elemento del ndarray è identificato dal nome della variabile seguito, tra parentesi quadre, dall'indice di posizione dell'elemento: l'indice di posizione è un solo numero intero per i vettori e corrisponde alla conta degli elementi da sinistra a destra partendo da zero; per le matrici è una tupla il cui primo elemento è l'indice di riga e il secondo è l'indice di colonna sapendo che l'angolo in alto a sinistra della matrice corrisponde a (0, 0).
Date le variabili appena create,

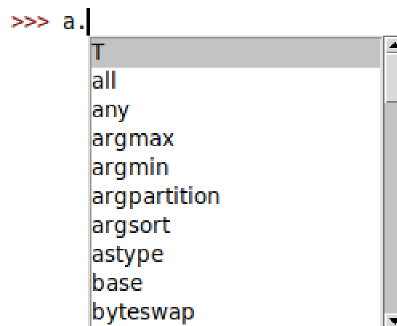
a[1] corrisponde a 2,
m[(1,2)] oppure semplicemente m[1,2] corrisponde a 7.

Ogni elemento del ndarray è a sua volta una variabile e ne può essere modificato il valore.
a[1] = 5 assegna al secondo elemento dell'array a il valore 5 e l'array a diventa 1 5 3.

Attraverso lo slicing possiamo identificare zone dell'array che sono a loro volta degli ndarray. Per un array x
x[2:4] identifica un array composto dal terzo e dal quarto elemento di un vettore (gli elementi compresi tra quello di indice 2 e quello di indice 4-1).
x[1,0:2] identifica un array composto dal primo e dal secondo elemento della seconda riga di una matrice.
x[2,2:] identifica un array composto da tutti gli elementi a partire dal terzo fino alla fine della terza riga di una matrice.

Se esploriamo il tipo della variabile che contiene l'array con la funzione type ci viene indicato che la variabile è di tipo ndarray.
type(m) restituisce <class 'numpy.ndarray'>.

Se nell'IDLE dove abbiamo creato un array assegnandolo ad una variabile digitiamo il nome di questa variabile seguito da un punto apriamo il lungo elenco di proprietà e metodi dell'oggetto ndarray



Agendo sulla barra di scorrimento sulla destra vediamo che l'elenco è molto lungo. Qui illustro solamente le proprietà e i metodi di uso più ricorrente e rimando alla documentazione ufficiale eventuali approfondimenti.

1.1.1 Proprietà

Possiamo esplorare alcune proprietà della variabile contenente ndarray attraverso le seguenti funzioni, richiamabili con la sintassi
<variabile>.<funzione>

dtype ritorna il tipo dei dati che formano l'array.
Con riferimento alle variabili create nel precedente paragrafo,
a.dtype ritorna dtype('int32').

ndim ritorna la dimensione dell'array.
Sempre con riferimento alle variabili create nel precedente paragrafo,
a.ndim ritorna 1,
m.ndim ritorna 2.

shape, applicata alle matrici, ritorna una tupla contenente il numero delle righe e il numero delle colonne
m.shape ritorna (2, 4).

1.1.2 Metodi

Esistono molti metodi di manipolazione e di estrazione di dati.

In generale questi metodi non alterano l'array su cui vengono applicati, in modo che il contenuto della variabile che li contiene non cambia. Per riutilizzare i risultati delle manipolazioni occorre creare nuove variabili che li contengano.

Unica eccezione, il seguente metodo

`sort()` ordina gli elementi dell'array.
Essendo un array `a` composto dagli elementi `1 5 3 2`
con `a.sort()`
esso diventa `1 2 3 5`

Tra i tanti altri metodi cito i seguenti, più comunemente utilizzati.

`copy()` crea una copia dell'array.
Si potrebbe pensare di fare una copia semplicemente inserendo il contenuto di una variabile in un'altra variabile: così facendo, tuttavia, una variazione apportata al contenuto agirebbe su entrambe le variabili.
Se creiamo la variabile copia con questa funzione ciò non accade.
Per esempio, tornando all'array `a` che abbiamo appena utilizzato, con `b = a` prima di applicare la funzione `sort()` avremo due array, `a` e `b`, contenenti gli elementi `1 5 3 2`.
Con `c = a.copy()` avremo anche l'array `c`, con gli stessi elementi `1 5 3 2`.
Dopo `a.sort()`, entrambe le variabili contenenti gli array `a` e `b` diventano `1 2 3 5`, mentre la variabile contenente l'array `c` rimane `1 5 3 2`.
Allo stesso modo, se ora modifichiamo il primo elemento dell'array `a` con `a[0] = 7`, gli array contenuti in `a` e `b` diventano `7 2 3 5` mentre l'array contenuto in `c` rimane sempre `1 5 3 2`.

`tolist()` converte l'array in una lista.
Se l'array è una matrice viene prodotta una lista di liste.
Dal momento che il ndarray ha dimensione fissa e non esistono suoi metodi per aggiungere elementi, la conversione in lista torna utile per poterlo fare con il metodo `append` dell'oggetto `list`.
Se, per esempio, abbiamo la seguente matrice `m`

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{vmatrix}$$

e vogliamo aggiungervi la riga `7 8 9`,
innanzi tutto convertiamo la matrice in una lista di liste con
`ml = m.tolist()`
aggiungiamo a questa lista la lista `[7,8,9]` con
`ml.append([7,8,9])`
e poi, utilizzando la nuova lista di liste, ricostruiamo la matrice m con
`m = np.array(ml)`, ottenendo così la nuova matrice `m`

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

`transpose()` genera la trasposta di una matrice.
Con
`mt = m.transpose()`
otteniamo la matrice `mt`, trasposta della matrice `m` creata nel precedente esempio

$$\begin{vmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{vmatrix}$$

`flatten()` trasforma una matrice in un vettore.
Data la matrice `m`

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$$

con `a = m.flatten()` otteniamo il vettore `a` con elementi `1 2 3 4`.

`reshape()` trasformiamo un vettore in una matrice.
Tra le partentesi tonde va inserito, come argomento, una tupla indicante il numero di righe e di colonne della matrice che si vuole ottenere.
Dato il vettore a dell'esempio precedente `1 2 3 4`, con `a.reshape((2,2))` oppure, semplicemente, `a.reshape(2,2)` otteniamo la matrice

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$$

Ovviamente il numero di elementi del vettore deve essere coerente con le dimensioni della matrice che vogliamo ottenere: da un vettore di 3 o di 5 elementi non potremo ottenere una matrice 2x2.

`min()` ritorna il valore minimo degli elementi che costituiscono un vettore o una matrice.

`max()` ritorna il valore massimo degli elementi che costituiscono un vettore o una matrice.

`mean()` ritorna la media del valore degli elementi che costituiscono un vettore o una matrice.

`std()` ritorna lo scarto quadratico medio tra il valore degli elementi che costituiscono un vettore o una matrice.

`sum()` ritorna la somma dei valori degli elementi che costituiscono un vettore o una matrice.

`prod()` ritorna il prodotto dei valori degli elementi che costituiscono un vettore o una matrice.

`cumsum()` ritorna un vettore i cui elementi sono la somma cumulativa dei valori degli elementi che costituiscono un vettore o una matrice.

Data la matrice `m`

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$$

`m.min()` ritorna `1`
`m.max()` ritorna `4`
`m.mean()` ritorna `2,5`
`m.std()` ritorna `1.1180339887498949`
`m.sum()` ritorna `10`
`m.prod()` ritorna `24`
`m.cumsum()` ritorna il vettore `1 3 6 10`.

Attraverso lo slicing possiamo riferire questi calcoli a zone di un vettore o di una matrice: `x[1,0:3].mean()` calcola la media tra i primi tre elementi della seconda riga di una matrice `x`.
Nel caso della nostra matrice `m`
`m[0,0:].mean()` ritorna `1,5`.

1.2 Operazioni matematiche

Inserendo gli operatori `+ - * / **` tra uno o più ndarray che abbiano le stesse dimensioni otteniamo un nuovo ndarray i cui elementi sono il risultato dell'applicazione dell'operatore elemento per elemento.

Dati i vettori

```
a = np.array((1,2,4))
b = np.array((3,1,2))
```

`a+b` ritorna un vettore composto da 4 3 6,
`a/b` ritorna un vettore composto da 0,3333 2 2,
`a**b` ritorna un vettore composto da `1 2 16`.

Date le matrici
`m = np.array(((2,3,1),(4,2,3)))`

$$\begin{vmatrix} 2 & 3 & 1 \\ 4 & 2 & 3 \end{vmatrix}$$

`n = np.array(((1,2,3),(3,2,2)))`

$$\begin{vmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \end{vmatrix}$$

`m-n` ritorna la matrice

$$\begin{vmatrix} 1 & 1 & -2 \\ 1 & 0 & 1 \end{vmatrix}$$

`m*n` ritorna la matrice

$$\begin{vmatrix} 2 & 6 & 3 \\ 12 & 4 & 6 \end{vmatrix}$$

che, attenzione, non è il prodotto matriciale ma il prodotto membro a membro.

Le stesse operazioni si possono fare tra un ndarray e uno scalare. In questo caso viene generato un nuovo ndarray i cui elementi sono costituiti dal risultato dell'operazione tra ciascun elemento e lo scalare.
Se prendiamo il nostro vettore `a` di prima
`a*3.5` ritorna `3,5 7 14`,
`a**2` ritorna `1 4 16`.
Se prendiamo la nostra matrice `n` di prima
`n/2` ritorna la matrice

$$\begin{vmatrix} 0.5 & 1 & 1.5 \\ 1.5 & 1 & 1 \end{vmatrix}$$

Allo stesso modo possiamo passare il ndarray come argomento ad una qualsiasi funzione simile a quelle contenute nel modulo math di Python di base, che il modulo NumPy replica in modo da renderla applicabile ad un ndarray (gli argomenti che possiamo passare ad una funzione del modulo math sono solo scalari). Otteniamo così un nuovo ndarray i cui elementi sono il risultato della funzione applicata a ciascun elemento.
Dato il vettore
`v = np.array((4,16,36,12))`
`np.sqrt(v)` ritorna `2 4 6 3,464`.

Per i nostri calcoli NumPy ci fornisce i valori a doppia precisione delle costanti matematiche π e e:
`np.pi` ritorna `3.141592653589793`,
`np.e` ritorna `2.718281828459045`.

1.3 Calcolo matriciale

Alla base del calcolo matriciale sta il prodotto scalare.

Il prodotto scalare tra due vettori aventi le stesse dimensioni corrisponde alla somma dei prodotti tra gli elementi dei vettori aventi lo stesso indice.
Se il vettore `a` è `1 2 3` e il vettore `b` è `3 5 0`
il prodotto scalare tra `a` e `b` è `(1*3)+(2*5)+(3*0) = 3+10+0 = 13`.

NumPy ha una funzione per calcolarlo, la funzione `dot()`.
Nel caso dei due vettori di cui sopra

`np.dot(a,b)` ritorna `13`.

Se abbiamo due matrici, di cui la prima ha un numero di colonne pari al numero di righe della seconda, attraverso il prodotto scalare calcoliamo il prodotto matriciale tra le due matrici, che è una matrice i cui elementi sono i prodotti scalari tra le righe della prima matrice e le colonne della seconda matrice.

Il prodotto matriciale tra `m` = $\begin{vmatrix} 1 & 2 \\ 5 & 4 \end{vmatrix}$ e `n` = $\begin{vmatrix} 2 & 6 \\ 1 & 3 \end{vmatrix}$ è una matrice che ha
all'indice [0,0] `(1*2)+(2*1) = 4`
all'indice [0,1] `(1*6)+(2*3) = 12`
all'indice [1,0] `(5*2)+(4*1) = 14`
all'indice [1,1] `(5*6)+(4*3) = 42`
cioè $\begin{vmatrix} 4 & 12 \\ 14 & 42 \end{vmatrix}$.

A questo risultato arriviamo passando alla funzione dot(), come argomenti, le due matrici `m` e `n`: `np.dot(m,n)`.

E qui si innesta l'algebra lineare.

NumPy ha il sotto-modulo `linalg` che contiene utili funzioni.

`inv()` calcola la matrice inversa.
Data la matrice `m` di prima,
`np.linalg.inv(m)` ritorna $\begin{vmatrix} -0.66666667 & 0.33333333 \\ 0.83333333 & -0.16666667 \end{vmatrix}$

`det()` calcola il determinante.
Sempre con la matrice m di prima,
`np.linalg.det(m)` ritorna `-6`

Ne avremmo abbastanza per risolvere sistemi di equazioni lineari, moltiplicando l'inversa della matrice dei coefficienti per il vettore dei termini noti, ma Numpy, con il sotto-modulo linalg, ci leva anche questo disturbo con la funzione `solve()`, passando alla quale la matrice dei coefficienti e il vettore dei termini noti otteniamo la soluzione del sistema.

Dato il sistema

$$\begin{cases} 2x - y = 2 \\ 3y + 2z = 16 \\ 5x + 3z = 21 \end{cases}$$

costruiamo la matrice dei coefficienti

```
m = np.array(((2,-1,0),(0,3,2),(5,0,3)))
```

e il vettore dei termini noti

```
n = np.array((2,16,21))
```

Possiamo ottenere il vettore della soluzione con

```
s = np.dot(np.linalg.inv(m),n)
```

oppure con

```
s = np.linalg.solve(m,n)
```

in entrambi i casi ottenendo il vettore `s = 3 4 2` dove `3` è il valore risolutivo per la `x`, `4` è il valore risolutivo per la `y` e `2` è il valore risolutivo per la `z`.

* * *

Se qualcuno vorrà approfondire i segreti di NumPy più di quanto abbia fatto io in questo rapido excursus troverà molto altro, soprattutto vedrà come esistano apparentemente molti modi di fare la stessa cosa.

Per esempio scoprirà che per costruire una matrice di tipo diverso da `ndarray` esiste la funzione `matrix()`, che genera un oggetto di tipo `matrix` che ha praticamente tutte le proprietà e i metodi di `ndarray` con la sola differenza che non può andare oltre le due dimensioni. Per cui non è del tutto vero che è la stessa cosa, anche, se per fare ciò che abbiamo visto finora, lo è.

2 Matplotlib

Matplotlib è una libreria Python per la visualizzazione di dati in modalità grafica.

All'indirizzo *https://matplotlib.org/* troviamo tutto ciò che riguarda Matplotlib e il suo utilizzo: documentazione, tutorial, ecc.

Prerequisito per l'utilizzo di Matplotlib è avere installato Python e il modulo Numpy.

Come per il modulo NumPy, l'installazione può facilmente avvenire con pip[3].

Con Matplotlib possiamo realizzare qualsiasi tipo di grafico e per descrivere le sue potenzialità servirebbe un intero volume.

Qui mi limito a descrivere l'applicazione di Matplotlib più vicina al calcolo scientifico, quella della visualizzazione di grafici di funzione.

Per fare questo si utilizza il sottomodulo `pyplot`, che si importa così
`import matplotlib.pyplot as plt`.

Le più importanti funzioni di questo sottomodulo sono
`plot()` per creare il grafico,
`title(<stringa>)` per dargli un titolo,
`grid()` per visualizzare una griglia,
`show()` per mostrare il grafico sullo schermo.

Gli argomenti della funzione `plot()` sono, nell'ordine:
la variabile contenente un array per l'asse della ascisse,
la variabile contenente l'espressione della funzione,
oltre, facoltativi:
`color = "<nome_colore>"` per indicare il colore del grafico,
`linewidth = "<numero> "` per indicare lo spessore della linea.
Se questi parametri facoltativi non sono indicati viene tracciato un grafico in blu con linea sottile (pari all'intero 1).

Prima dell'eventuale utilizzo della funzione `show()` possiamo utilizzare la funzione
`savefig("<path_e_nome_file>.png")` per salvare il grafico in formato `.png`.
Cosa che possiamo fare anche utilizzando un pulsante contenuto nella finestra del grafico mostrata dalla funzione `show()`.

Il seguente script traccia il grafico della funzione $y = x^2$ tra -5 e 5, in colore verde, con il titolo «Grafico di funzione», con griglia visibile e lo memorizza in un file in formato .png e lo visualizza sullo schermo.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5,5,50)
y = x**2
plt.plot(x, y, color = "green")
plt.title("Grafico di funzione")
plt.grid()
plt.savefig("/home/vittorio/Immagini/grafico.png")
plt.show()
```

Il terzo argomento passato alla funzione `linspace()` di NumPy per creare l'array della variabile indipendente deve essere un numero abbastanza elevato in modo da creare una bella curva continua nel grafico.

Se vogliamo dare un titolo agli assi abbiamo a disposizione le funzioni
`xlabel(<stringa>)` per dare un titolo all'asse delle ascisse,
`ylabel(<stringa>)` per dare un titolo all'asse delle ordinate.

[3] v. nota 1 a pagina 3

3 SciPy

SciPy sta per Scientific Python ed è una collezione di funzioni e algoritmi matematici usati in campo scentifico, costruita su NumPy.

All'indirizzo *https://scipy.org/* troviamo una serie di link a tutte le librerie Python utilizzabili in campo scientifico, comprese le due viste finora. Scegliendo il link SCIPY LIBRARY troviamo tutto ciò che riguarda SciPy e il suo utilizzo: documentazione, tutorial, ecc.

Prerequisito per l'utilizzo di SciPy è avere installato Python e il modulo NumPy.

Come per gli altri moduli, l'installazione può facilmente avvenire con pip[4].

La raccolta delle funzioni è organizzata in più sotto-moduli:
`cluster` contiene algoritmi di clustering,
`constants` contiene costanti matematiche e fisiche,
`fftpack` contiene funzioni della Trasformata di Fourier,
`integrate` per integrazione ed equazioni differenziali ordinarie,
`interpolate` per interpolazione e smoothing spline,
`io` per Input e Output,
`linalg` contiene funzioni di algebra lineare,
`ndimage` per image processing n-dimensionale,
`odr` per la regressione delle distanze ortogonali,
`optimize` contiene funzioni di ottimizzazione e di ricerca di radici,
`signal` per elaborazione di immagini,
`sparse` per matrici sparse e funzioni associate,
`spatial` per strutture dati spaziali e algoritmi,
`special` contiene altre funzioni speciali,
`stats` per distribuzioni statistiche e funzioni correlate.

Per usare SciPy occorre importarlo e il modo migliore per farlo è quello di limitare l'importazione al sotto-modulo che serve con la sintassi
`import scipy.`<sottomodulo> `as` <sigla>.
Con
```
import scipy.constants as c
```
ci teniamo a disposizione il sotto-modulo `constants` e possiamo richiamarne le proprietà e le funzioni con la sintassi
`c.`<proprietà/funzione>.

Della quindicina di sotto-moduli che compongono SciPy propongo qui un'introduzione ai sei di più ricorrente utilizzo.

Per un panorama completo e più approfondito rimando alla SciPy Reference Guide che possiamo consultare e scaricare in formato PDF dal sito web di SciPy.

3.1 Costanti

Il sotto-modulo constants contiene costanti matematiche e fisiche.

Le costanti matematiche, in realtà, sono solo due: pi greco (π) e rapporto aureo (ϕ).
Avendo importato il sotto-modulo `constants` con la sigla `c`,
`c.pi` ritorna `3.141592653589793`,
`c.golden` ritorna `1.618033988749895`.

Moltissime le costanti del mondo della fisica, per il cui elenco completo rimando alla Reference Guide.
Come esempio cito
`c.g` che ritorna `9.80665`, la costante di gravitazione universale,
`c.Avogadro` che ritorna `6.022140857e+23`, la costante di Avogadro.

[4]v. nota 1 a pagina 3

3.2 Algebra lineare

Il sotto-modulo `linalg` contiene praticamente le stesse funzioni dell'omonimo sottomodulo di `numpy` che abbiamo visto prima:
`inv()` calcola la matrice inversa,
`det()` calcola il determinante,
`solve(A,b)` calcola il valore delle incognite in un sistema lineare,
e rimando alle esemplificazioni fatte prima.

Altra funzione utile, pure questa contenuta anche nel sotto-modulo `linalg` di `numpy` ma di cui non ho parlato è la funzione `lstsq(A,b)` che calcola il valore delle incognite in un sistema lineare non compatibile con il metodo dei minimi quadrati.

Sappiamo che per risolvere in maniera esatta un sistema di equazioni lineari occorre che esso sia compatibile, cioè che il numero delle equazioni sia uguale al numero delle incognite. In questo modo la matrice dei coefficienti è una matrice quadrata e se ne può calcolare l'inversa per applicare il metodo risolutivo $A^{-1}b = x$ oppure se ne può calcolare il determinante per applicare il metodo risolutivo di Cramer. In ogni caso, trovate le soluzioni x, si verifica che $b - Ax = 0$. Ed è ciò che avviene utilizzando la funzione `solve()`.

Con la funzione `lstsq()` possiamo risolvere in maniera approssimata un sistema di equazioni non compatibili e le soluzioni saranno tali per cui $b - Ax$ non è uguale a zero ma è del valore minimo possibile secondo il principio dei minimi quadrati.

Importato NumPy con

```
import numpy as np
```

e il sotto-modulo `linalg` di SciPy con

```
import scipy.linalg as ln
```

e dato il sistema

$$\begin{cases} 2x + 3y - 5z = 5 \\ 3x + 2y + 2z = 2 \end{cases}$$

costruiamo la matrice dei coefficienti con

```
A = np.array(((2,3,-5),(3,2,2)))
```

e il vettore dei termini noti con

```
b = np.array((5,2)),
```

infine, con

```
ln.lstsq(A,b)
```

otteniamo una lista il cui primo elemento è l'array

```
array([ 0.56074766, 0.58411215, -0.42523364])
```

che elenca, nell'ordine, i valori delle incognite `x`, `y` e `z`.
La lista contiene altre cose che lasciamo ai matematici professionisti.

3.3 Interpolazione

L'interpolazione è il processo con cui valutiamo il più probabile valore ignoto tra due valori noti.

Il sotto-modulo `interpolate` di SciPy offre parecchie funzioni per attuare questo processo. Qui illustro la più semplice: `interp1d`, nelle due varianti opzionali lineare e cubica.

Supponiamo di avere otto misurazioni della temperatura di una giornata a determinate ore:

ora	gradi
3	14
5	13
7	14
10	16
12	18
14	19
18	17
23	16

Importiamo il modulo numpy con
import numpy as np,

importiamo il sotto-modulo interpolate di scipy con
import scipy.interpolate as int.

Creiamo gli array dei dati denominando `x` l'array delle ore e l'array `y` quello delle temperature:

```
x = np.array((3,5,7,10,12,14,18,23)),
y = np.array((14,13,14,16,18,19,17,16)).
```

Troviamo la linea interpolante con
f1 = int.interp1d(x, y, kind = 'linear') per l'interpolazione lineare,
f2 = int.interp1d(x, y, kind = 'cubic') per l'interpolazione cubica.

Se non si utilizza l'opzione kind viene effettuata l'interpolazione lineare.

Se vogliamo conoscere la più probabile temperatura alle ore 16, ora per la quale non esiste la rilevazione, passiamo questo valore come argomento alla funzione interpolante:
`f1(16)` ritorna `18`,
`f2(16)` ritorna `18.426358355165736`
a dimostrazione del fatto che la dinamica dell'andamento della temperatura desunta dalle osservazioni è diversa a seconda del tipo di linea interpolante scelta.

Possiamo avere contezza grafica di tutto ciò.

Importiamo il plotter di Matplotlib con

```
import matplotlib.pyplot as plt.
```

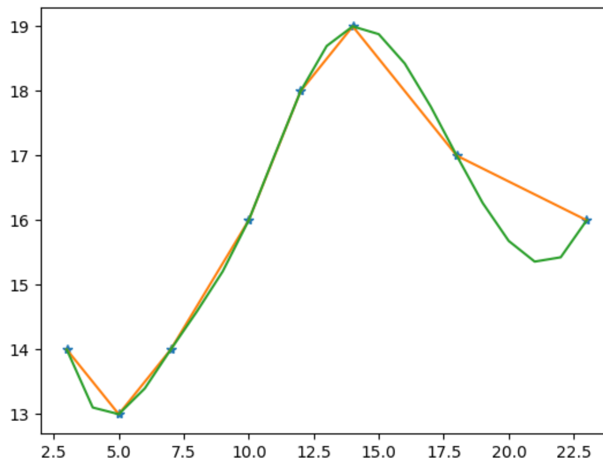
Creiamo un nuovo array per l'asse delle ascisse scandito su tutte le ore del nostro intervallo di misurazione con:

```
xnew = np.linspace(3,23,21).
```

Ora costruiamo un grafico che mostri i dati osservati, la linea di interpolazione lineare f1 e la curva di interpolazione cubica f2 con:

```
plt.plot(x,y,'*',xnew,f1(xnew),xnew,f2(xnew))
```

e con `plt.show()` lo mostriamo



I valori osservati sono contrassegnati da un asterisco (notare la sintassi con cui lo si è scelto con l'opzione '*').

La linea arancione corrisponde all'interpolante lineare.

La curva verde corrisponde all'interpolante cubica.

* * *

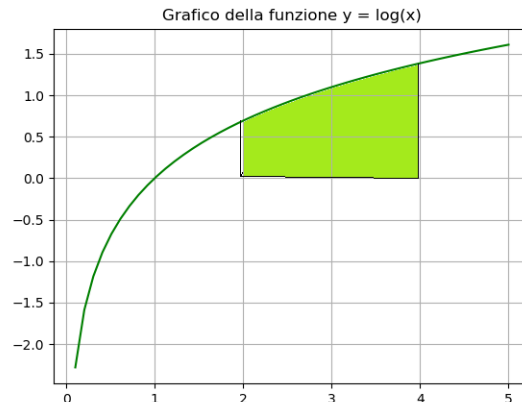
Qui ho presentato la più semplice applicazione del sotto-modulo `interpolate` di SciPy attraverso la funzione `interp1d` che lavora su due variabili.

Se siamo in presenza di tre variabili abbiamo a disposizione la funzione `interp2d`.

Ma vi sono tante altre funzioni per le quali rimando alla Reference Guide.

3.4 Integrazione

Nel calcolo numerico l'integrazione consiste praticamente nel determinare l'area compresa tra un segmento dell'asse delle ascisse e la corrispondente linea del grafico di una funzione: la così detta integrazione definita.



In questo esempio, che riguarda la funzione $y = log(x)$, essa corrisponde alla zona colorata di verde e si esprime in questo modo

$$\int_2^4 log(x)\, dx$$

Il sotto-pacchetto `integrate` contiene parecchie funzioni per il calcolo integrale (integrali semplici, doppi, tripli, a quadratura gaussiana, ecc.) e rimando alla Reference Guide per approfondimenti. Qui mostro l'uso della funzione più semplice, quella che calcola l'area di cui abbiamo parlato, che si chiama `quad()`.

Importiamo il sotto-pacchetto con

```
import scipy.integrate as i
```

Importiamo anche il modulo `math` di Python per utilizzarne la funzione `log()`

```
import math
```

Definiamo la funzione di cui calcolare l'integrale definito con

```
def f(x):
    return math.log(x)
```

e, per stare all'esempio da cui siamo partiti, ne calcoliamo l'integrale tra 2 e 4 così

```
i.quad(f, 2, 4).
```

Il risultato è la seguente tupla

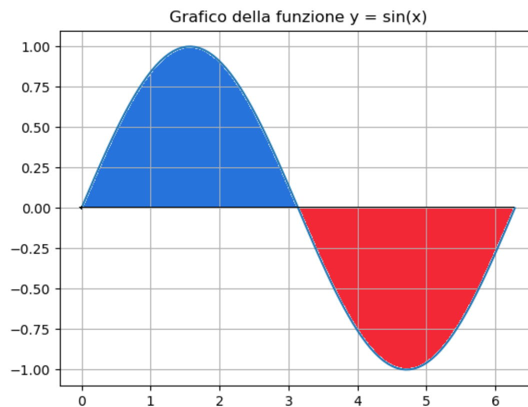
```
(2.158883083359672, 2.3968417066196 59e-14)
```

il cui primo membro è il ricercato valore dell'area e il secondo, come si vede un numero molto piccolo, indica l'errore di approssimazione del calcolo.

Sugli integrali definiti va ricordato che l'area calcolata in corrispondenza a valori negativi della funzione è negativa e, se calcoliamo l'integrale per tratti in cui la funzione è in parte positiva e in parte negativa, l'area negativa va a sottrarsi a quella positiva.

Se vogliamo calcolare l'area effettiva, comprendente sia la parte positiva sia la parte negativa, dobbiamo effettuare il calcolo delle aree separatamente per le zone con segno diverso della funzione, sommando ai valori positivi il valore assoluto dei valori negativi.

Prendiamo, per esempio, la funzione $y = sin(x)$



L'area contrassegnata in colore blu, corrispondente a valori positivi della y, ha valore positivo mentre quella contrassegnata in colore rosso, corrispondente a valori negativi della y, ha valore negativo.
Dal momento che le due aree sono uguali, se calcoliamo l'integrale definito della funzione nell'intervallo tra 0 e 2π che le comprende entrambe, il risultato è zero.

Pertanto, se vogliamo sapere l'area effettiva delle due zone messe insieme possiamo procedere così.

Importiamo ciò che ci serve con

```
import scipy.integrate as i
import numpy as np
```

L'importazione di `numpy` anziché di `math` ci serve per la funzione che calcola il valore assoluto. Dal momento che il risultato da indicare come argomento della funzione `absolute()`, presente in entrambi i moduli, è una tupla, la funzione del modulo `math` non va bene in quanto accetta come argomento un solo valore. Da che ci siamo utilizziamo anche la funzione `sin()` di NumPy.

Definiamo la funzione con

```
def f(x):
   return np.sin(x)
```

Ora, con

```
i.quad(f,0,2*np.pi)
```

otteniamo l'area `2.2579663352318637e-16`
mentre con

```
i.quad(f,0,np.pi) + np.absolute(i.quad(f,np.pi,2*np.pi))
```

otteniamo l'area `4.00000000e+00`.
Praticamente 0 nel primo caso e 4 nel secondo.

Precisazione finale.

Negli esempi ho utilizzato le funzioni `log()` e `sin()`, che sono funzioni predefinite nei moduli Python, ma la funzione da integrare possiamo anche costruircela noi.

Per esempio, per calcolare

$$\int_0^3 x^2 - 2x + 1\,dx$$

costruiamo la funzione con

```
def f(x):
   return x**2-2*x+1
```

e con

```
i.quad(f,0,3)
```

otteniamo il risultato
`(2.9999999999999996, 3.330669073875469e-14)`.

3.5 Ottimizzazione

In matematica l'ottimizzazione consiste nella ricerca degli input per una funzione obiettivo ai quali corrisponda l'output minimo o massimo della funzione stessa.

Il sotto-pacchetto `optimize` contiene moltissime funzioni per la ricerca dei minimi di una funzione. Qui mostro l'uso delle funzioni più semplici e rimando alla Reference Guide per approfondimenti.

Il fatto che gli algoritmi siano tutti finalizzati alla minimizzazione non ci deve scoraggiare se abbiamo come obiettivo la ricerca di un massimo: basta, infatti, che anteponiamo segno negativo ai valori restituiti dalla funzione obiettivo e la ricerca del minimo si converte nella ricerca di un massimo.

Le semplici funzioni che utilizziamo come principianti sono:
`minimize_scalar()` per la ricerca del minimo assoluto,
`minimize()` per la ricerca di tutti i minimi, compresi quelli relativi.

Nel primo caso basta che passiamo alla funzione, come argomento, la funzione obiettivo. Nel secondo caso dobbiamo passare come argomenti, oltre alla funzione obiettivo, separato da virgola, un valore tentativo che avvii la ricerca iterativa.

Vediamo come funziona.

Sia questa la nostra funzione obiettivo

$$x^2 + 8\sin(x)$$

il cui grafico è il seguente

Importiamo il sotto-pacchetto con
```
import scipy.optimize as opt
```
Importiamo anche il modulo `math` di Python per utilizzarne la funzione `sin()`
```
import math
```
Definiamo la funzione obiettivo con
```
def f(x):
   return x**2+8*math.sin(x)
```
e con
```
opt.minimize_scalar(f)
```
veniamo informati del fatto che la ricerca ha avuto successo e che il minimo assoluto della funzione è
```
-6.0294035242910322
```
corrispondente al valore della x di
```
-1.2523532340863732
```
Allo stesso risultato perveniamo con
```
opt.minimize(f,-3)
```

avviando l'iterazione da sinistra con il valore tentativo -3.
Se ci portiamo verso destra, per esempio con il valore tentativo 4,
```
opt.minimize(f,4)
```
scopriamo che ha avuto successo la ricerca di un altro minimo, questa volta relativo, con valore della funzione
```
9.41977563692598
```
corrispondente al valore della x di
```
3.59530513
```
Per ricercare i massimi della funzione, basta che anteponiamo il segno - alla funzione obiettivo definendola così
```
def f(x):
   return -(x**2+8*math.sin(x))
```
e con
```
opt.minimize_scalar(f)
```
veniamo informati del fatto che la ricerca ha avuto successo e che esiste un massimo con valore della funzione
`11.318349214841467` (rendendo positivo il risultato)
corrispondente al valore della x di
```
2.1333322446613039
```
Il sotto-modello `optimize` contiene anche funzioni per la ricerca delle radici della nostra funzione obiettivo, cioè dei punti in cui il grafico della funzione si incrocia con l'asse delle ascisse, i così detti zeri della funzione. La più semplice è
```
fsolve()
```
alla quale dobbiamo passare come parametri la funzione obiettivo e, separato da virgola, un valore tentativo per avviare l'iterazione.

Riferendoci alla funzione obiettivo dei precedenti esempi, con
```
opt.fsolve(f, -10)
```
avviando l'iterazione da sinistra, troviamo una prima radice per x uguale a
```
-2.36630242
```
e con
```
opt.fsolve(f, -1)
```
procedendo verso destra, troviamo una seconda radice per x uguale a
```
0
```
e, corrispondendo la nostra funzione ad una equazione di secondo grado, non ci saranno altre radici.

3.6 Statistica

Già tra i metodi del `ndarray` di NumPy abbiamo visto essere presenti alcune funzioni di statistica descrittiva (valore minimo, valore massimo, media dei valori e scarto quadratico medio).

Ma per la statistica abbiamo il sotto-modulo di SciPy `stats`, che contiene un gran numero di strumenti per descrivere campioni statistici, per lavorare con distribuzioni di probabilità e per eseguire diverse tipologie di test statistici.

Possiamo importarlo con la formula sin qui usata
```
import scipy.stats as st.
```
Dal momento che tutti i dati su cui lavoriamo devono essere nel formato `ndarray`, dobbiamo sempre importare anche NumPy con
```
import numpy as np.
```
Con un'unica funzione, `describe()`, produciamo in blocco alcuni indicatori di statistica descrittiva.
Dato l'array
```
a = np.array((2,4,6.5,12,7,6,8,15.8))
```
con
```
st.describe(a)
```

produciamo una tupla contenente il numero degli elementi (8), i valori minimo e massimo (2.0 e 15.8), la media (7.6625), la varianza (19.31125), l'indice di asimmetria (0.68989) e l'indice di curtosi (-0.392214).

Per un dilettante potrebbe bastare, ma c'è molto di più e servirebbe un intero libro per parlarne. Se nell'IDLE di Python, avendo importato il sotto-pacchetto come st scriviamo st seguito da un punto (st.) possiamo scorrere il lungo elenco delle funzioni disponibili e lo statistico professionista può apprezzarne l'abbondanza.

Sempre a livello dilettantesco possiamo costruire questi due array

```
x = np.array((34,45,58,62,78,112,148))
y = np.array((78,89,112,123,138,190,220))
```

e trovare la correlazione esistente tra di loro calcolando l'indice r di Pearson con la funzione

```
st.pearsonr(x,y)
```

che genera la tupla

```
(0.99335214818166018, 6.8961823195309017e-06
```

dove il primo elemento è l'indice r e il secondo è il possibile errore di determinazione.

Con gli stessi array possiamo determinare la retta di regressione con la funzione

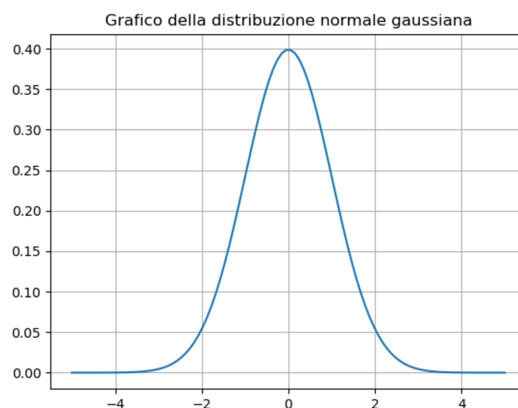
```
st.linregress(x,y)
```

che genera una tupla contenente il coefficiente angolare (slope), l'intercetta, il coefficiente di determinazione e i possibili errori.

Come esempio di funzioni relative a distribuzioni di probabilità cito il più classico, che ha a che fare con la curva gaussiana, detta anche normale.
Per crearla abbiamo a disposizione la funzione norm() che, utilizzata senza indicare alcun argomento, crea la normale vera e propria. Tra le parentesi possiamo indicare due argomenti per personalizzare una gaussiana: il primo per indicare il valore medio della distribuzione e il secondo, separato da virgola, per indicare la deviazione standard della distribuzione.
Con il seguente script

```
import scipy.stats as st
import numpy as np
import matplotlib.pyplot as plt
y = st.norm()
x = np.linspace(-5, 5, 100)
px = y.pdf(x)
plt.plot(x, px)
plt.title("Grafico della distribuzione normale gaussiana")
plt.grid()
plt.show()
```

otteniamo il grafico della gaussiana



Il passaggio fondamentale è quello con cui viene costruito l'oggetto «distribuzione normale» con

`y = st.norm()`
Tutto il resto riguarda la costruzione del grafico utilizzando una variabile `x` da mettere sulle ascisse e la funzione, chiamata `px`, che rappresenta la densità di probabilità corrispondente ai valori della `x`.
La densità di probabilità si ottiene con il metodo `pdf()` (probability density function) dell'oggetto `y`.
Altri importanti metodi dell'oggetto `y` sono
`mean()` che fornisce la media della distribuzione
nel caso della normale classica `y.mean()` ritorna `0`;
`std()` che fornisce la deviazione standard
nel caso della normale classica `y.std()` ritorna `1` (il σ);
`cdf()` che fornisce la probabilità cumulativa (cumulative distribution function).
Quest'ultima funzione è molto utile in quanto per differenza tra la probabilità cumulativa di un valore superiore e la probabilità cumulativa di un valore inferiore si ottiene la probabilità che un valore sia compreso in un intervallo: in questo modo si ottengono i valori leggibili nella tavola della funzione $\Theta(\lambda)$. Per differenza tra 1 e questa probabilità si ottiene la probabilità che un valore sia fuori dall'intervallo.
Così, sempre utilizzando l'oggetto `y` che rappresenta la distribuzione normale,
`y.cdf(1)-y.cdf(-1)` restituisce `0.68268949213708585`
probabilità corrispondente a σ
`y.cdf(2)-y.cdf(-2)` restituisce `0.95449973610364158`
probabilità corrispondente a 2σ
`y.cdf(3)-y.cdf(-3)` restituisce `0.99730020393673979`
probabilità corrispondente a 3σ
`y.cdf(4)-y.cdf(-4)` restituisce `0.99993665751633376`
probabilità corrispondente a 4σ, la perfezione.
Con questi metodi possiamo determinare la probabilità che si verifichi un certo dato in presenza di una qualsiasi distribuzione per la quale siano noti la media e la deviazione standard (scarto quadratico medio).
Poniamo, per esempio, di essere in presenza di una serie di dati la cui media sia 5 e lo scarto quadratico medio sia 2.
Costruiamo la distribuzione normale caratterizzata da questi dati con
d = st.norm(5,2).
Se vogliamo conoscere la probabilità che la distribuzione presenti dati con scarto dalla media inferiore a 1 facciamo la differenza
`d.cdf(6)-d.cdf(4)` e troviamo il risultato `0.38292492254802624`
che rappresenta la probabilità che si riscontrino dati compresi tra 4 e 6
e da cui deduciamo che dati inferiori a `4` o superiori a `6` si potranno riscontrare con probabilità `0.6170750774519738` (complemento a `1` della precedente probabilità).

Per finire questa carrellata esemplificativa e tutt'altro che esauriente vediamo un test statistico, il test χ^2 di Pearson per valutare se la differenza tra due distribuzioni statistiche sia accidentale o significativa.
Abbiamo tre distribuzioni in altrettanti ndarray:

```
a = np.array((38,45,36,47,58,42,35))
b = np.array((42,51,40,61,87,68,48))
c = np.array((39,45,37,45,59,43,36))
```

Con
`st.chisquare(a,b)`
otteniamo il valore del test, `27.8286`, e la relativa probabilità, `0.000101`, che, essendo inferiore al valore critico di `0,05` indica che la differenza tra la distribuzione `a` e la distribuzione `b` è significativa.
Con
`st.chisquare(a,c)`

otteniamo il valore del test, `0.2095`, e la relativa probabilità, `0.9998`, che, essendo superiore al valore critico di `0,05` indica che la differenza tra la distribuzione `a` e la distribuzione `c` è accidentale.

* * *

A chi voglia avere un'idea di cosa offra Python per il calcolo scientifico posso dire che ciò che ho illustrato in questo manualetto penso corrisponda si e no ad un 25%.