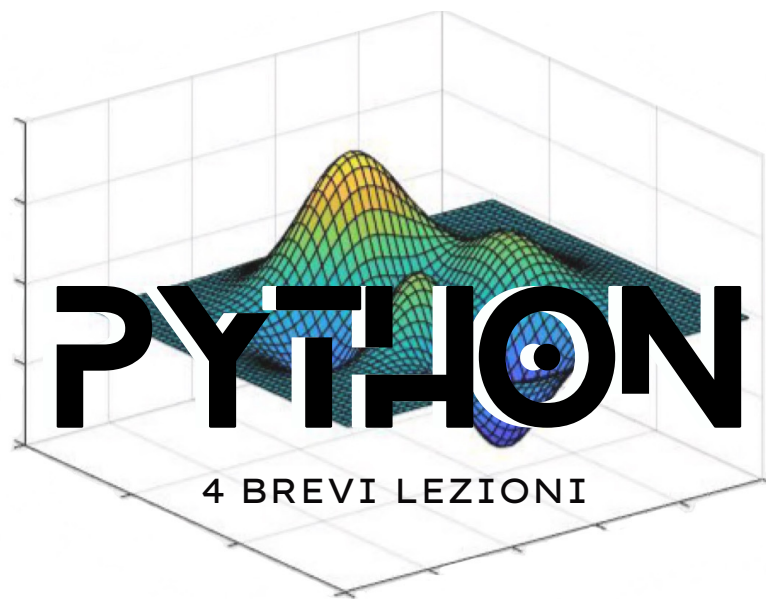


4 Brevi lezioni di Python

Francesco Zeno Costanzo

(Do you remember the) 21(st night of) September 2024



I think, it's time we blow this scene.
Get everybody and the stuff together.
Ok three, two, one, let's jam.
Seatbelts, Tank! (1999)

Indice

1	Esercizi seconda lezione	3
1.1	Soluzioni	3
1.2	Esercizi aggiuntivi	4
2	Terza lezione	6
2.1	Le funzioni	6
2.2	Istruzioni di controllo	6
2.2.1	Espressioni condizionali: if, else, elif	6
2.2.2	Cicli: while, for	7
2.3	Ancora funzioni	9
2.4	Funzioni lambda	9
2.5	Grafici	10
2.6	Standard input	13
2.7	Prestazioni: <i>pure</i> Python vs librerie	14
2.8	Gestione errori	15
2.8.1	Try e except	15
2.8.2	Raise Exception	16
2.9	Esercizi	16

1 Esercizi seconda lezione

Vi riporto la soluzione degli esercizi dati alla scorsa lezione, di cui riporto comunque il testo qua sotto:

1. Verificare che anche con le liste l'operazione di assegnazione fa coincidere gli indirizzi di memoria.
2. Creare inizialmente una matrice L 4×4 tutta nulla ma la cui diagonale sia: "-1, 1, 1, 1", e poi creare una matrice B definita come

$$B = \begin{bmatrix} \gamma & -\beta\gamma & 0 & 0 \\ -\beta\gamma & \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{con} \quad \gamma = 1/\sqrt{1-\beta^2} \quad \beta \in [0, 1].$$

preso v un vettore a vostra scelta consa succede se calcolate $v^T L v$ e $w^T L w$, con $w = Bv$? Confrontare tra l'utilizzo di L e quello di I (ovviamente 4×4), per vari valori di β .

3. Data una matrice di zeri 4×2 (4 righe, 2 colonne), riempite la colonna di sinistra e poi invertite le colonne.
4. Dato un `linspace` (e.g. un array che contiene tutti i numeri tra 0 e 20) creare due nuovi array con solo i numeri pari uno e solo i dispari l'altro.
5. Creare una matrice che contenga sulle colonne le tabelline da 0 a 10 (non banale da fare in poche righe).
6. Creare un `logspace` in base 10 senza usare l'apposita funzione di `numpy`.
7. Create una matrice 3×3 con numeri a caso ma reali compresi fra 0 e 1 e stampateli prima normalmente e poi con solo due cifre decimali (cercate su internet, non morde).
8. Prendete un array a caso non ordinato di 10 elementi e sostituite il valore più grande con 7 e il più piccolo con $1/7$.
9. Dato un array contenente valori ripetuti sempre creato da voi (ormai siete bravissimi a creare array dato che lo lascio sempre a voi) creare un array che contenga tutti gli elementi senza ripetizione.

1.1 Soluzioni

Ecco la mia soluzione di questi esercizi. non riporto l'output per brevità.

Primo:

```
1 a = [1, 2, 3, 4]
2 print(f"array iniziale: {a}, id: {id(a)}")
3
4 b = a
5 b[0] = 7
6
7 print(f"array iniziale: {a}, id: {id(a)}")
8 print(f"array finale : {b}, id: {id(b)}")
```

Secondo:

```
1 import numpy as np
2
3 L = np.zeros((4, 4))
4 I = np.zeros((4, 4))
5 B = np.zeros((4, 4))
6 idx = [0, 1, 2, 3] # lista degli indici
7
8 L[idx, idx] = [-1, 1, 1, 1]
9 I = np.eye(4)
10
11 beta = 0.1
12 gamma = 1/np.sqrt(1 - beta**2)
13 B[idx, idx] = [gamma, gamma, 1, 1]
14 B[0, 1] = B[1, 0] = -beta*gamma
15
16 v = np.array([4, 8, 2, 1]) # vettore a caso
17 print(v.T @ L @ v, v.T @ I @ v)
18 w = B @ v
19 print(w.T @ L @ w, w.T @ I @ w)
```

Terzo:

```
1 import numpy as np
2
3 A = np.zeros((4, 2))
4 A[:, 0] = 1
5 print(A)
6 # faccio lo swap cambiando l'ordine degli indici
7 print(A[:, [1, 0]])
```

Quarto:

```
1 import numpy as np
2
3 x = np.linspace(0, 20, 21, dtype=int)
4
5 pari = x[x%2 == 0] # maschera per i pari
6 disp = x[x%2 == 1] # maschera per i dispari
7 print(pari)
8 print(disp)
```

Quinto:

```
1 import numpy as np
2
3 x = np.linspace(0, 10, 11, dtype=int)
4 # trasformo il vettore in una matrice N x 1
5 y = x[:, None]
6 print(x)
7 print(y)
8 print(x*y)
```

Sesto:

```
1 import numpy as np
2
3 decade_iniziale = -3 # da 10^-3
4 decade_finale = 3 # fino a 10^3
5 base = 10 # base della scala
6 numero_punti = 15
7
8 x = np.linspace(decade_iniziale, decade_finale, numero_punti)
9 x_log = base**x
10 n_log = np.logspace(decade_iniziale, decade_finale, numero_punti)
11
12 print(x_log)
13 print(n_log)
```

Settimo:

```
1 import numpy as np
2
3 # Ogni entrata della matrice e' una variabile estratta da una distribuzione uniforme
4 Mat = np.random.random((3, 3))
5 print(Mat)
6
7 np.set_printoptions(precision=2)
8 print(Mat)
```

Ottavo:

```
1 import numpy as np
2
3 x = np.random.random(10)
4 print(x)
5 x[x.argmin()], x[x.argmax()] = 1/7, 7
6 print(x)
```

Nono:

```
1 import numpy as np
2
3 x = np.array([1, 2, 2, 3, 4, 4, 5])
4 print(x)
5 unique_x = np.unique(x)
6 print(unique_x)
```

1.2 Esercizi aggiuntivi

Siccome la scorsa lezione non avevamo ancora introdotto gli operatori logici e il prodotto matriciale, vi lascio altri esercizi che ora siete in grado di risolvere (la soluzione naturalmente verrà messa nelle dispense della prossima lezione):

1. Creare la matrice identità $I_{3 \times 3}$ e verificare che, dato un vettore v a vostra scelta, $v^T I v$ sia uguale a "sum($v**2$)".
2. Dato un array a vostro piacimento, creare una maschera, con due condizioni sempre a vostro gusto.
3. Dati due array, createli come volete, dovete eliminare dal primo array, tutti gli elementi che esso condivide con il secondo array.

4. Dato l'array $x=(1, 2, 3)$, visto come insieme, creare una matrice (9×2) che sia il prodotto cartesiano di x con se stesso (i.e. tutte le possibili coppie che potete formare).
5. Prendete due vettori lunghi tre che rappresentino la posizione e la velocità di un oggetto puntiforme, calcolate il momento angolare.
6. Dato un array 2D, la posizione di un particella su un piano, in cartesiane passare alla scrittura in coordinate polari.

2 Terza lezione

2.1 Le funzioni

Don't repeat yourself: è questa la logica delle funzioni. Le funzioni sono frammenti di codici, atti a ripetere sempre lo stesso tipo di operazioni con diversi valori dei parametri in input a seconda delle esigenze. Come al solito vediamo degli esempi:

```
1 def area(a, b):
2     """
3     restituisce l'area del rettangolo
4     di lati a e b
5     """
6     A = a*b #calcolo dell'area
7     return A
8
9 #chiamiamo la funzione e stampiamo subito il risultato
10 print(area(3, 4))
11 print(area(2, 5))
12
13 """
14 Se la funzione non restituisce nulla
15 ma esegue solo un pezzo di codice,
16 si parla propriamente di procedura
17 e il valore restituito e' None.
18 """
19 def procedura(a):
20     a = a+1
21
22 print(procedura(2))
23
24 """
25 Volendo si possono creare anche funzioni
26 che non hanno valori in ingresso:
27 """
28 def pigreco():
29     return 3.14
30 print(pigreco())
31
32 [Output]
33 12
34 10
35 None
36 3.14
```

Portiamo all'attenzione due fatti importanti:

- È fondamentale in Python che il corpo della funzione sia indentato, per seguire un raggruppamento logico del codice. Lo stesso vale per altri costrutti che vedremo tra poco. Insomma le parti indentate del codice devono essere logicamente connesse.
- Definendo degli argomenti per una funzione si creano delle variabili "locali", il cui nome non influenza tutto quello che c'è fuori dalla funzione stessa. Ad esempio, per la funzione area abbiamo definito una variabile "A", ma posso tranquillamente definire una nuova variabile "A" al di fuori della funzione e non avrei problemi di sovrascrittura.

Abbiamo visto che le funzioni possono prendere dei parametri o anche nessun parametro, quindi la domanda che sorge spontanea è: ne possono prendere infiniti? La risposta è sì ma prima di vederlo facciamo una piccola deviazione e parliamo delle istruzioni di controllo.

2.2 Istruzioni di controllo

Per istruzioni di controllo si intendono dei comandi che modificano il flusso di compilazione di un programma in base a determinati confronti e/o controlli su certe variabili. Ci sono casi in cui il computer deve fare cose diverse a seconda degli input o fare la stessa cosa un certo numero di volte fino a che una certa condizione sia o non sia soddisfatta.

2.2.1 Espressioni condizionali: if, else, elif

Tramite l'istruzione if effettuiamo un confronto/controllo. Se il risultato è vero il programma esegue la porzione di codice immediatamente sotto-indentata. In caso contrario, l'istruzione else prende il controllo e il programma esegue la porzione di codice indentata sotto quest'ultima. Se l'istruzione else non è presente e il controllo

avvenuto con l'if risultasse falso, il programma semplicemente non fa niente. Vediamo il caso classico del valore assoluto:

```
1 def assoluto(x):
2     """
3     restituisce il valore assoluto di un numero
4     """
5     # se vero restituisci x
6     if x >= 0:
7         return x
8     #altrimenti restituisci -x
9     else:
10        return -x
11
12 print(assoluto(3))
13 print(assoluto(-3))
14
15 [Output]
16 3
17 3
```

È possibile aggiungere delle coppie if/else in cascata tramite il comando "elif", che è identico semanticamente a "else if"; per esempio:

```
1 def segno(x):
2     """
3     funzione per capire il segno di un numero
4     """
5     #se vero ....
6     if x > 0:
7         return 'Positivo'
8     #se invece ....
9     elif x == 0:
10        return 'Nullo'
11    #altrimenti ....
12    else:
13        return 'Negativo'
14
15 print(segno(5))
16 print(segno(0))
17 print(segno(-4))
18
19 [Output]
20 Positivo
21 Nullo
22 Negativo
```

2.2.2 Cicli: while, for

Partiamo con in cicli while: essi sono porzioni di codice che iterano le stesse operazioni fino a che una certa condizione risulta essere verifica:

```
1 def fattoriale(n):
2     """
3     Restituisce il fattoriale di un numero
4     """
5     R = 1
6     #finche' e' vero fai ...
7     while n > 1:
8         R *= n
9         n -= 1
10    return R
11
12 print(fattoriale(5))
13
14 [Output]
15 120
```

Un'accortezza da porre con i cicli while è verificare che effettivamente la condizione inserita si verifichi altrimenti il ciclo non si interrompe e va avanti per sempre, ed è molto molto tempo, della serie che fa prima a decadere il protone.

Passando ai cicli for invece essi ripetono una certa azione finché un contatore non raggiunge il massimo. Vediamo come implementare il fattoriale con questo ciclo:

```

1 def fattoriale(n):
2     """
3     restituisce il fattoriale di un numero
4     """
5     R = 1
6     #finche' i non arriva ad n fai ...
7     for i in range(1, n+1):
8         R = R*i
9     return R
10
11 print(fattoriale(5))
12
13 [Output]
14 120

```

Abbiamo quindi introdotto una variabile ausiliaria "i" utilizzata in questo contesto come contatore, cioè come variabile che tiene il conto del numero di cicli effettuati. Nel caso in esame, stiamo dicendo tramite l'istruzione for che la variabile "i" deve variare all'interno della lista $\text{range}(1, n+1) = [1, 2, \dots, n]$. Il programma effettua l'operazione $R = R*i$ per tutti i valori possibili che i assume in questa lista, nell'ordine. Da notare il comando range che crea una lista sulla quale iterare, ma noi abbiamo visto già le liste e gli array e abbiamo visto che presentano alcune somiglianze, un'altra somiglianza da far vedere è che entrambi sono 'iterabili' e quindi possiamo iterarci sopra:

```

1 import numpy as np
2
3 def trova_pari(array):
4     """
5     restituisce un array contenente solo
6     i numeri pari dell'array di partenza
7     """
8     R = np.array([]) #array da riempire
9     #per ogni elemento in array fai ...
10    for elem in array:
11        if elem%2 == 0:
12            R = np.append(R, elem)
13    return R
14
15 a = np.array([i for i in range(0, 11)])
16 """
17 il precedente e' un modo piu' conciso di scrivere:
18 a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
19 """
20 print(a)
21 print(trova_pari(a))
22
23 [Output]
24 [ 0  1  2  3  4  5  6  7  8  9 10]
25 [ 0.  2.  4.  6.  8. 10.]

```

In questo esempio abbiamo utilizzato gli array ma si potrebbe senza problemi rifare tutto con le liste. Altri due comandi interessanti per quanto riguarda i cicli sono: enumerate e zip.

enumerate:

```

1 import numpy as np
2
3 #creiamo un array
4 array = np.linspace(0, 1, 5)
5
6 """
7 in questo modo posso iterare contemporaneamente
8 sia sugli indici sia sugli elementi dell'array
9 """
10 for index, elem in enumerate(array):
11     print(index, elem)
12
13 [Output]
14 0 0.0
15 1 0.25
16 2 0.5
17 3 0.75
18 4 1.0

```

zip:

```

1 import numpy as np
2

```



```

3 #creiamo tre un array
4 array1 = np.linspace(0, 1, 5)
5 array2 = np.linspace(1, 2, 5)
6 array3 = np.linspace(2, 3, 5)
7 """
8 in questo modo posso iterare contemporaneamente
9 sugli elementi di tutti gli array
10 """
11 for a1, a2, a3 in zip(array1, array2, array3):
12     print(a1, a2, a3)
13
14 [Output]
15 0.0 1.0 2.0
16 0.25 1.25 2.25
17 0.5 1.5 2.5
18 0.75 1.75 2.75
19 1.0 2.0 3.0

```

Anche qui come le funzioni è necessario indentare.

2.3 Ancora funzioni

Dopo questa digressione torniamo alle funzioni, abbiamo detto che una funzione può prendere infiniti argomenti, ma dal punto di vista pratico come lo implementiamo, in un modo semi decente? Una risposta sarebbe quella di passare alla funzione non delle singole variabili ma un array o una lista, cosa che si può fare tranquillamente, e lavorare poi all'interno della funzione con gli indici per utilizzare i vari elementi dell'array, o della lista, o ciclarci sopra. Un altro modo per farlo è usare: *args (args è un nome di default, potremmo chiamarlo mimmo):

```

1 def molt(*numeri):
2     """
3     restituisce il prodotto di n numeri
4     """
5     R = 1
6     for numero in numeri:
7         R *= numero
8     return R
9
10 print(molt(2, 7, 10, 11, 42))
11 print(molt(5, 5))
12 print(molt(10, 10, 2))
13
14 [Output]
15 64680
16 25
17 200

```

L'esempio appena visto non è altro che la funzione fattoriale di prima leggermente modificata e che non prende più in input una sequenza crescente di numeri. I parametri vengono passati come una tupla e in questo caso il simbolo "*" viene definito operatore di unpacking proprio perché "spacchetta" tutte le variabili che vengono passate alla funzione.

2.4 Funzioni lambda

Esiste un particolare tipo di funzioni in Python, le cosiddette funzioni lambda, sono fondamentalmente uguali alle funzioni dichiarate dall'utente ma devono avere una singola espressione, quindi niente calcoli complicati nel mezzo. La sintassi è: lambda argomenti : espressione. Sono in genere usate come anonime, magari da passare ad altre funzioni ad esempio se bisogna fittare con una retta, (vedere lezione successiva) posso usare curve_fit così: curve_fit(lambda x, m, q : x*m + q, x, y, ...); ma ciò non vieta comunque di dargli un nome.

```

1 f = lambda x : 3*x
2 print(f(2))
3
4 h = lambda x, y, z : x*y + z
5 print(h(2, 3, 4))
6
7 [Output]
8 6
9 10

```

Ovviamente esse possono anche essere usate all'interno di una funzione magari come ciò che la funzione ritorna, il che vuol dire che la funzione restituirà un'altra funzione.

```

1 def g(x):
2     '''restituisce potenza x-esima di y
3     '''
4     return lambda y : y**x
5
6 G = g(3) # potenza cubica
7 print(G(4))
8
9 #=====
10 #=====
11
12 def m(f, y, x):
13     '''passo una funzione ad una funzione
14     '''
15     return f(y) - x
16
17 print(m(lambda x : x**2, 5, 1))
18
19 [Output]
20 64
21 24

```

2.5 Grafici

Fare un grafico è un modo pratico e comodo di visualizzare dei dati, qualsiasi sia la loro provenienza. Capita spesso che i dati siano su dei file (per i nostri scopi in genere file .txt o .csv) e che i file siano organizzati a colonne:

```

1 #t[s] x[m]
2 1      1
3 2      4
4 3      9
5 4     16
6 5     25
7 6     36

```

Per leggerli:

```

1 import numpy as np
2
3 #Leggiamo da un file di testo classico
4 path = 'dati.txt'
5 dati1, dati2 = np.loadtxt(path, unpack=True)
6 """
7 unpack=True serve proprio a dire che vogliamo che
8 dati1 contenga la prima colonna e dati2 la seconda
9 La prima riga avendo il cancelletto verra' saltata
10 """
11
12 #se vogliamo invece che venga letto tutto come una matrice scriviamo:
13 path = 'dati.txt'
14 dati = np.loadtxt(path) # sarebbe unpack = False
15 #dati sara' nella fattispecie una matrice con due colonne e 6 righe
16
17
18 #leggere da file.csv
19 path = 'dati.csv'
20 dati1, dati2 = np.loadtxt(path, usecols=[0,1], skiprows=1, delimiter=',', unpack=True)
21 """
22 si capisce senza troppa fatica che usecols indica le colonne che vogliamo leggere
23 skiprows il numero di righe da saltare e delimiter indica il carattere che separa le colonne
24 """

```

Un interessante tipo di file sono i file con estensione ".npy" comodi se i dati sono di dimensioni elevate:

```

1 import time
2 import numpy as np
3
4 x = np.linspace(0, 1, int(2e6)) # dati
5
6 #=====
7 # file txt
8 #=====
9
10 # salviamo su file
11 start = time.time()

```

```

12 path = r'c:\Users\franc\Desktop\dati.txt'
13 f = open(path, 'w')
14 for i in x:
15     f.write(f'{i} \n')
16 f.close()
17 end = time.time() - start
18
19 print(f"tempo di scrittura txt: {end} s")
20
21 #legghiamo da file
22 start = time.time()
23 X = np.loadtxt(path, unpack=True)
24 end = time.time() - start
25
26 print(f"tempo di lettura txt: {end} s")
27
28 #=====
29 # file npy
30 #=====
31
32 # salviamo su file
33 start = time.time()
34 path = r'c:\Users\franc\Desktop\dati.npy'
35 np.save(path, x)
36 end = time.time() - start
37
38 print(f"tempo di scrittura npy: {end} s")
39
40 #legghiamo da file
41 start = time.time()
42 X = np.load(path, allow_pickle='TRUE')
43 end = time.time() - start
44
45 print(f"tempo di lettura npy: {end} s")
46
47 [Output]
48 tempo di scrittura txt: 5.610752582550049 s
49 tempo di lettura txt: 9.489601373672485 s
50 tempo di scrittura npy: 0.016022205352783203 s
51 tempo di lettura npy: 0.015637636184692383 s

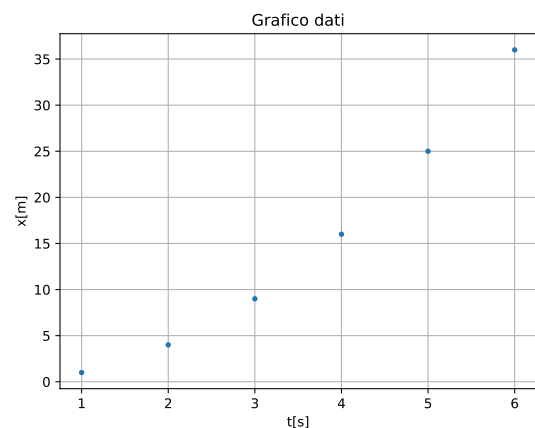
```

Abbiamo usato la libreria time per misurare il tempo, "time.time()" restituisce i numeri di secondi trascorsi dall'inizio dell'epoca unix (1 gennaio 1970). Come vediamo la differenza di tempi è abissale. Passiamo ora alla creazione di un grafico. Creare ora un grafico è semplice grazie all'utilizzo della libreria matplotlib:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #Leggiamo da un file di testo classico
5 path = 'dati.txt'
6 dati1, dati2 = np.loadtxt(path, unpack=True)
7
8 plt.figure(1) #creiamo la figura
9
10 #titolo
11 plt.title('Grafico dati')
12 #nomi degli assi
13 plt.xlabel('t[s]')
14 plt.ylabel('x[m]')
15 #plot dei dati
16 plt.plot(dati1, dati2, marker='.', linestyle='')
17 #aggiungiamo una griglia
18 plt.grid()
19 #comando per mostrare a schermo il grafico
20 plt.show()

```



Commentiamo un attimo quanto fatto: dopo aver letto i dati abbiamo fatto il grafico mettendo sull'asse delle ascisse la colonna del tempo e su quello delle ordinate la colonna dello spazio; se all'interno del comando "plt.plot(...)" scambiassimo l'ordine di dati1 e dati2 all'ora gli assi si invertirebbero, non avremmo più x(t) ma t(x). Inoltre il comando "marker='.'" sta a significare che il simbolo che rappresenta il dato deve essere un punto; mentre il comando "linestyle=''" significa che non vogliamo che i punti siano uniti da una linea (linestyle='-' dà una linea, linestyle='- -' dà una linea tratteggiata).

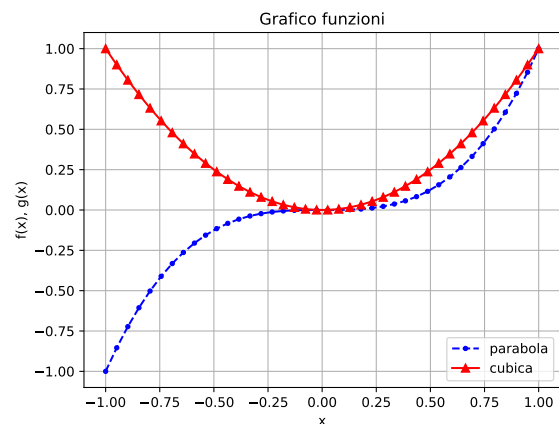
Se invece volessimo graficare una funzione o più definite da codice? Anche qui i comandi sono analoghi:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x):
5     """
6     restituisce il cubo di un numero
7     """
8     return x**3
9
10 def g(x):
11     """
12     restituisce il quadrato di un numero
13     """
14     return x**2
15
16 #array di numeri equispaziati nel range [-1,1] usiamo:
17 x = np.linspace(-1, 1, 40)
18
19 plt.figure(1) #creiamo la figura
20
21 #titolo
22 plt.title('Grafico funzioni')
23 #nomi degli assi
24 plt.xlabel('x')
25 plt.ylabel('f(x), g(x)')
26 #plot dei dati
27 plt.plot(x, f(x), marker='.', linestyle='--', color='blue', label='parabola')
28 plt.plot(x, g(x), marker='^', linestyle='-', color='red', label='cubica')
29 #aggiungiamo una leggenda
30 plt.legend(loc='best')
31 #aggiungiamo una griglia
32 plt.grid()
33 #comando per mostrare a schermo il grafico
34 plt.show()

```

Notare che per distinguere le due funzioni oltre al "marker" e al "linestyle" abbiamo aggiunto il comando "color" per dare un colore e il comando "label" che assegna un'etichetta poi visibile nella legenda (loc='best' indica che Python la mette dove ritiene più consono, in modo che non rischi magari di coprire porzioni di grafico). Ovviamente è consigliata una lettura della documentazione per conoscere tutti gli altri comandi possibili per migliorare/abbellire il grafico da adde alle funzioni già presenti. Altre funzioni utili possono essere: "plt.axis(...)" che imposta il range da visualizzare su entrambi gli assi; il comando "plt.xscale(...)" che permette di fare i grafici con una scala, magari logaritmica o altro sull'asse x (analogo sarà sulle y mutatis mutandis).



Ultima menzione da fare sono gli istogrammi:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 plt.figure(1)
5 plt.title('grafico a barre')
6 plt.xlabel('valore')
7 plt.ylabel('conteggi')
8 # Sull'asse x utilizziamo un array di 10 punti equispaziati.
9 x = np.linspace(1,10,10)
10 # Sull'asse y abbiamo, ad esempio, il seguente set di dati:
11 y = np.array([2.54, 4.78, 1.13, 3.68, 5.79, 7.80, 5.4, 3.7, 9.0, 6.6])
12
13 # Il comando per la creazione dell'istogramma corrispondente e':
14 plt.bar(x, y, align='center')
15
16 plt.figure(2)
17 plt.title('istogramma di una distribuzione gaussiana')
18 plt.xlabel('x')
19 plt.ylabel('p(x)')

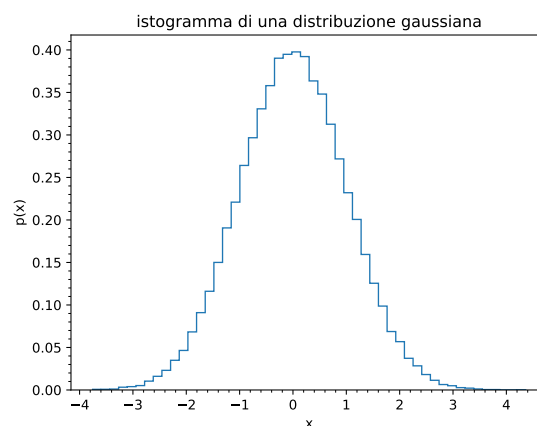
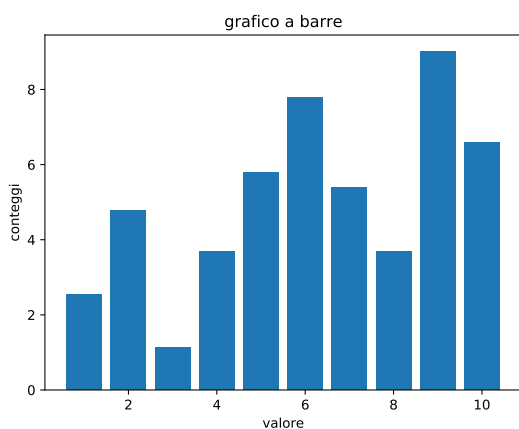
```

```

20
21 """
22 lista di numeri distribuiti gaussianamente con media 0 e varianza 1
23 si usa l'underscore nel for poiche' non serve usare
24 un'altra variabile. Avremmo potuto scrivere for i ...
25 ma la i non sarebbe comparsa da nessun'altra parte
26 sarebbe stato uno spreco
27 """
28 z = [np.random.normal(0, 1) for _ in range(int(1e5))]
29 plt.hist(z, bins=50, density=True, histtype='step')
30 plt.minorticks_on() # tick piccoli sugli assi
31
32 plt.show()

```

Piccolo appunto che bisogna fare, nel caso di "plt.hist()" bisogna stare attenti perché il numero di bin va scelto con cura (qui abbiamo scritto 50 sulla fiducia). Se tale parametro viene fornito come un intero (pari a n per esempio), suddivide i nostri dati in n barre equispaziate¹, tuttavia se gli viene passata una sequenza (una lista o, equivalentemente, una tupla) oppure come un altro tipo di dato (per esempio come stringa) presenta dei comportamenti differenti: invitiamo dunque il lettore a leggere la documentazione relativa a "matplotlib.pyplot.hist".



2.6 Standard input

È (a proposito, la È si fa premendo Alt+0200 sul tastierino, quantomeno su windows) inoltre possibile dare al codice che abbiamo scritto degli input da shell. Esistono due modi per farlo: l'uso della funzione "input()", oppure utilizzare "argparse" per dare al codice ciò che serve direttamente da linea di comando su shell, ad esempio se fossimo su una partizione linux senza un editor stile pyzo. Cominciamo con la prima possibilità:

```

1 """
2 Programma per calcolare il trinagolo di tartaglia
3 """
4 import numpy as np
5
6 # leggo da input un valore e lo rendo intero
7 # la stringa verra stampata su shell
8 n = int(input("Ordine del triangolo: "))
9 a = np.zeros((n, n), dtype=int) # matrice per i coefficienti
10
11 # calocolo i coefficienti del trinagolo
12 a[0,0] = 1
13 for i in range(1, n):
14     a[i, 0] = 1
15     for j in range(1, i):
16         a[i, j] = a[i-1, j-1] + a[i-1, j]
17     a[i,i] = 1
18
19 # stampo a schermo
20 for i in range(n):
21     for j in range(i+1):
22         # solo per fare la forma a piramide
23         if j == 0 : # non funziona con numeri a due cifre
24             print(*[" "]*(n-i),a[i, j], end='')
25         else:

```

¹Cosa accade se avete meno dati dei bins inseriti? Banalmente si creano delle colonne vuote

```

26         print("",a[i, j], end='')
27     print()
28
29 [Output]
30 Ordine del triangolo: 5
31     1
32   1 1
33 1 2 1
34 1 3 3 1
35 1 4 6 4 1

```

Vediamo ora come usare argparse. Ora però il codice è più comodo eseguirlo su una shell, che sia quella di anaconda o quella della vostra distro linux è uguale. Le modifiche al codice sono veramente poche:

```

1 """
2 Programma per calcolare il trinagolo di tartaglia
3 """
4 import argparse
5 import numpy as np
6
7 description='Programma per calcolare il trinagolo di tartaglia leggendo le informazioni da
8   linea di comando'
9 # descrizione accessibile con -h su shell
10 parser = argparse.ArgumentParser(description=description)
11 parser.add_argument('dim', help='Dimensione della matrice, ovvero potenza del binomio')
12 args = parser.parse_args()
13 n = int(args.dim) # accedo alla variabile tramite il nome messo a linea 10
14
15 a = np.zeros((n, n), dtype=int) # matrice per i coefficienti
16
17 # calcolo i coefficienti del trinagolo
18 a[0,0] = 1
19 for i in range(1, n):
20     a[i, 0] = 1
21     for j in range(1, i):
22         a[i, j] = a[i-1, j-1] + a[i-1, j]
23     a[i,i] = 1
24
25 # stampra a schermo
26 for i in range(n):
27     for j in range(i+1):
28         # solo per fare la forma a piramide
29         if j == 0 : # non funziona con numeri a due cifre
30             print(*[" "]*(n-i),a[i, j], end='')
31         else:
32             print("",a[i, j], end='')
33     print()

```

Vi metto uno screen della shell per capire cosa è successo (un po' sgranata ma pazienza):

```

C:\Windows\System32\Windc  +  -  x
(base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codicil3> python tartaglia_argparse.py -h
usage: tartaglia_argparse.py [-h] dim

Programma per calcolare il trinagolo di tartaglia leggendo le informazioni da linea di comando

positional arguments:
  dim                Dimensione della matrice, ovvero potenza del binomio

optional arguments:
  -h, --help          show this help message and exit
(base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codicil3> python tartaglia_argparse.py 5
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
(base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codicil3>

```

2.7 Prestazioni: *pure* Python vs librerie

Avevamo accennato al fatto che Python fosse lento ma che utilizzando le librerie si potesse un po' migliorare le prestazioni, vediamo un esempio:

```

1 import time
2 import numpy as np
3
4 #inizio a misurare il tempo
5 start = time.time()

```

```

6
7
8 a1 = 0          #variabile che conterra' il risultato
9 N = int(5e6)   # numero di iterazioni da fare = 5 x 10**6
10
11 #faccio il conto a 'mano'
12 for i in range(N):
13     a1 += np.sqrt(i)
14
15 #finisco di misurare il tempo
16 end = time.time()-start
17
18 print(end)
19
20 #inizio a misurare il tempo
21 start = time.time()
22
23 #stesso conto ma fatto tramite le librerie di python
24 a2 = sum(np.sqrt(np.arange(N)))
25
26 #finisco di misurare il tempo
27 end = time.time()-start
28
29 #sperabilmente sara' minore del tempo impiegato prima
30 print(end)
31
32 [Output]
33 11.588378429412842
34 0.8475463390350342

```

Vediamo che quindi usando le funzioni di numpy, (np.arange) e le funzioni della libreria standard di Python (sum), è possibile fare lo stesso conto in un tempo molto minore che tramite un ciclo for. Questo perché le librerie non sono totalmente in Python ma in molta parte in C e/o fortran.

2.8 Gestione errori

È molto facile scrivere codice che produca errore, magari perché distrattamente ci siamo dimenticati qualcosa o magari qualcosa è stato implementato male. Esiste un costrutto che ci permette di gestire gli errori in maniera tranquilla diciamo. Facciamo un semplice esempio:

```

1 a = 0
2 b = 1/a
3 print(b)
4
5 [Output]
6 Traceback (most recent call last):
7   File "<tmp 1>", line 5, in <module>
8     b = 1/a
9 ZeroDivisionError: division by zero

```

Abbiamo fatto una cosa molto brutta, nemmeno Dio può dividere per zero (al più possiamo appellarci alla censura cosmica e mettere un'orizzonte a vestire la divisione per zero) e quindi il computer ci dà errore. Possiamo aggirare il problema, evitando così il second impact, in due modi diciamo:

2.8.1 Try e except

Possiamo utilizzare il costrutto try except dicendo al computer: prova a fare la divisione e, sia mai funziona, se questa però dà errore, e l'errore è "ZeroDivisionError" allora assegna a b un altro valore. In questo modo eventuali istruzioni presenti dopo vengono eseguite e il codice non si arresta.

```

1 a = 0
2 try :
3     b = 1/a
4 except ZeroDivisionError:
5     b = 1
6
7 print(b)
8
9 [Output]
10 1

```

Anche qui è fondamentale indentare il blocco delle istruzioni.

2.8.2 Raise Exception

Mettiamo il caso in cui ci siano operazioni da fare in cui il valore della variabile "b" è importante, quindi sarebbe meglio interrompere il flusso del codice perché con un dato valore il risultato finale sarebbe poco sensato. Si può fare il controllo del valore e sollevare un'eccezione per fermare il codice.

```
1 """
2 leggo un valore da shell
3 uso del comando try per evitare che venga letto
4 qualcosa che non sia un numero: e.g. una stringa
5 """
6 try:
7     b = int(input('scegliere un valore:'))
8 except ValueError:
9     print('hai digitato qualcosa diverso da un numero, per favore ridigitare')
10    b = int(input('scegliere un valore:'))
11
12 #se si sbaglia a digitare di nuovo il codice si arresta per ValueError
13
14 #controllo se e' possibile proseguire
15 if b > 7 :
16     #se vero si blocca il codice sollevando l'eccezione
17     messaggio_di_errore = 'il valore scelto risulta insensato in quanto nulla supera 7, misura
18     massima di ogni cosa'
19     raise Exception(messaggio_di_errore)
20
21 [Output]
22 8
23 Traceback (most recent call last):
24   File "<tmp 1>", line 18, in <module>
25     raise Exception(messaggio_di_errore)
26 Exception: il valore scelto risulta insensato in quanto nulla supera 7, misura massima di ogni
27     cosa
```

2.9 Esercizi

Anche qui voglio lasciarvi qualche esercizio per farvi prendere confidenza con quanto imparato finora. Ci saranno però delle modifiche. Infatti di alcuni esercizi non vi darò la soluzione ma solo un file che testa la vostra proposta di soluzione. Tale test vi darà un paio di info sul vostro codice, ad esempio la memoria che esso utilizza, il tempo che impiega, ed anche implementato il test tramite pylint. Si tratta di un pacchetto molto interessante che analizza la sintassi del vostro codice secondo quelle che sono le regole canoniche di scrittura, vi evidenzia gli errori e vi dà una valutazione. Mi raccomando se nel testo dell'esercizio vi dico di chiamare una funzione in un certo modo, fatelo, così il codice test può fare i confronti del caso. Tale file di test sarà un codice python chiamato "test.py" che potete eseguire normalmente evi si aprirà la finestra. I più coraggiosi posso cimentarsi nel leggere e capire tale codice volendo "*audentes fortuna iuvat*". Vi faccio vedere sotto, alla soluzione del primo esercizio, come appare la finestra del test.

1. Scrivere una funzione "area(l, n)" che calcoli l'area di un poligono regolare di lato "l" e numero di lati "n". Scrivere poi una funzione "pitagora(lista_lati, n)" che prenda in input una lista di tre elementi (una terna pitagorica) e un numero di lati "n" e che restituisca la somma delle aree delle figure sui cateti a cui sottraete l'area della figura sull'ipotenusa. Fatelo magari per vari valori. Di seguito una formula che potrebbe tornarvi utile:

$$A = \frac{1}{2}(nl) \frac{l \cot(\pi/n)}{2}$$

2. Fare con un ciclo più plot su uno stesso grafico, dove la funzione deve dipendere dalla variabile su cui si cicla (e.g. x^i con x un array di un certo range e i la variabile del ciclo).
3. Stessa cosa di sopra ma ora ogni curva deve avere un colore e uno linestyle diverso e una legenda.
4. Creare una funzione che legga da input un numero intero con la condizione che esso sia maggiore di zero e che dia la possibilità di inserirlo nuovamente finché la condizione non è verificata.
5. Rifare il primo esercizio usando una funzione lambda
6. Sovrapporre i plot di un istogramma e della funzione di distribuzione associata, a vostra scelta, e aggiungere al grafico tutte le bellurie del caso.
7. Scrivere una funzione "osservabile(data)" che dato un array "data" ne restituisca la media e la deviazione standard in un array.

$$\mu = \sum_i^n x_i \quad \sigma = \sqrt{\frac{1}{n(n-1)} \sum_i^n (x_i - \mu)^2}$$

8. (righe, colonne e Sudoku!). Questo esercizio è guidato a "scaletta" ed è volto a costruire qualche funzione utile che potrebbero essere utilizzate per risolvere un Sudoku.
- (a) Presa una matrice 9×9 costruire una funzione che prende in ingresso un numero i (che rappresenta la riga in cui ci troviamo) e un intero $1 \leq n \leq 9$ e verifica che n non sia già presente sulla riga. *Hint*: bisogna fare un ciclo su...
 - (b) Presa una matrice 9×9 costruire una funzione che prende in ingresso un numero j (che rappresenta la colonna in cui ci troviamo) e un intero $1 \leq n \leq 9$ che verifica che n non sia già presente sulla riga. *Hint*: come prima, ma va fatto su...
 - (c) (difficile) Vogliamo adesso costruire una funzione che, preso in ingresso un numero $1 \leq n \leq 9$ e una possibile posizione che vogliamo dare (all'interno del Sudoku) ad n data nella forma i, j che indicano rispettivamente la riga e la colonna in cui si sta cercando di metterlo, verificare che la cella del Sudoku dove vogliamo mettere n non ce l'abbia già al suo interno.
Hint: la parte più difficile dell'esercizio, se vi siete cimentati, è sicuramente trovare i bordi della cella (a meno che fare caso per caso) in una maniera elegante (ovvero senza fare caso per caso). Se ci pensate bene, potete sempre trovare la riga x e la colonna y che identificano l'angolo in alto a sinistra della cella in cui ci troviamo approssimando per difetto una divisione e moltiplicandolo per lo stesso numero che avete diviso (pensate a come è fatto il Sudoku e a quante celle ci sono lungo una riga/colonna)
 - (d) Costruire una funzione che prende in ingresso un intero $1 \leq n \leq 9$ e una posizione i, j (rispettivamente riga e colonna della matrice) che verifica se un numero può essere inserito o meno all'interno della cella presa (tramite un **True** o **False**).
Hint: bisogna rimettere insieme quanto scritto nei precedenti punti

See you Space Cowboy ...