

6 Terza lezione

6.1 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.
- 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 b, ma posso tranquillamente definire una nuova variabile b al di fuori della funzione.

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.

6.2 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.

6.2.1 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

```

6.2.2 Cicli: while, for

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

```

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.

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.

6.3 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.

6.4 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, unpack=True)
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 a differenza di quanto sopra dobbiamo specificare le colonne da usare (contiamo da zero)
23 a causa poi dell'organizzazione dei .csv dobbiamo dire anche quante righe saltare
24 """

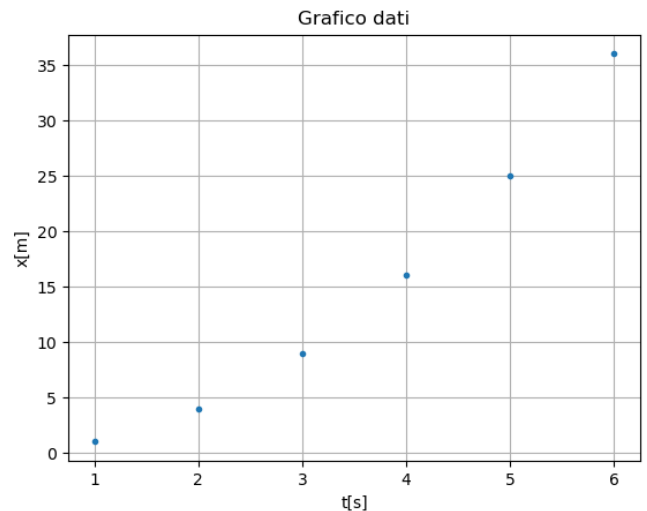
```

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')

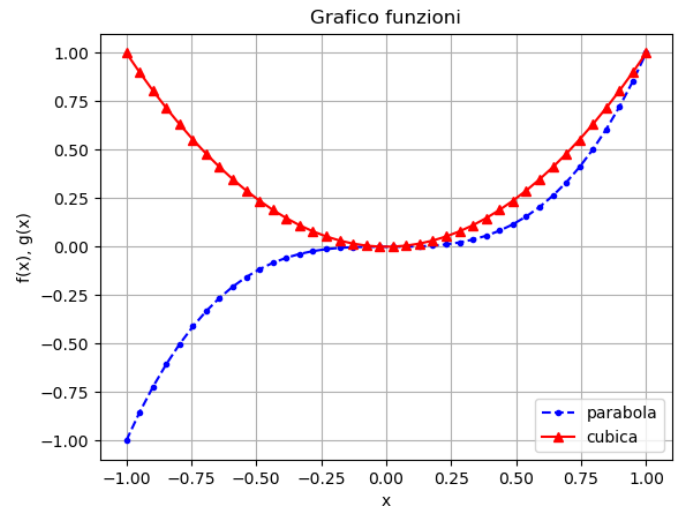
```

```

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).



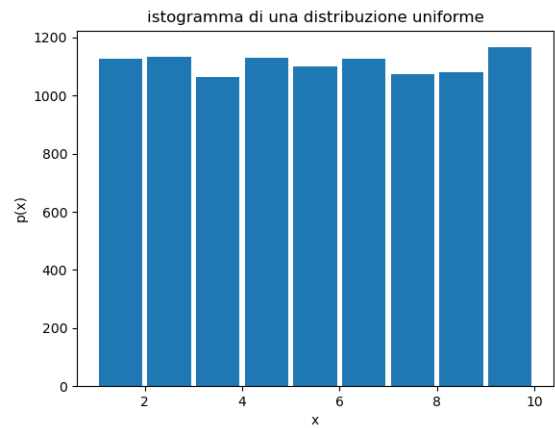
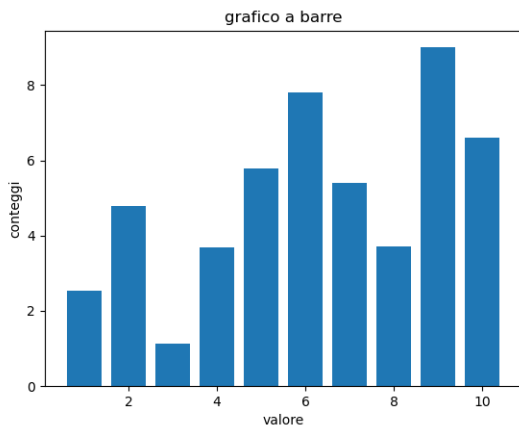
Ultima menzione da fare sono gli istogrammi (il fetish dei pallettari):

```

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 uniforme')
18 plt.xlabel('x')
19 plt.ylabel('p(x)')
20
21 """
22 lista di numeri distribuiti uniformemente fra 0 e 10
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.uniform(10) for _ in range(10000)]
29 plt.hist(z, bins=10, rwidth=0.9)
30
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 nove sulla fiducia).



6.5 Esercizio riassuntivo

Vogliamo provare a fare un breve esercizio che riassume quanto fatto, proviamo a calcolare le aree di tre poligoni regolari di lati rispettivamente 3, 4, 5 e numero di lato generici. Questo scrivendo una funzione "Area(l, n)" per poi confrontare il valore di "Area(3, 4) + Area(4, n)" con quello di "Area(5, n)" per ogni valore di n.

```

1 import numpy as np
2
3 def Area(l, n):
4     """
5     calcolo area di un poligono
6     regolare di lato l e numero lati n
7     """
8     a = 1/2 * l/np.tan(np.pi/n) #apotema
9     p = n*l                      #perimetro
10    A = p*a/2                    #area
11    return A
12
13 l = [3, 4, 5] #dimensioni dei lati, deve essere una terna pitagorica
14 n = np.arange(4, 12) #numero di lati dei poligoni
15
16 A3 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 3
17 A4 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 4
18 A5 = np.array([]) #array in cui ci saranno le aree dei poligoni di lato 5
19
20 for nn in n: #loop sul numero di lati
21     for ll in l: #lup sulla dimensione, quindi sul triangolo
22         A0 = Area(ll, nn) #calcolo dell'area
23
24         if ll == 3:
25             A3 = np.append(A3, A0)
26         elif ll == 4:
27             A4 = np.append(A4, A0)
28         elif ll == 5:
29             A5 = np.append(A5, A0)
30
31
32 for i in range(len(n)):
33     print(f'A3 + A4 = {A3[i]+A4[i]:.3f}')
34     print(f'A5 = {A5[i]:.3f}')
35
36 [Output]
37 A3 + A4 = 25.000
38 A5 = 25.000
39 A3 + A4 = 43.012
40 A5 = 43.012
41 A3 + A4 = 64.952
42 A5 = 64.952
43 A3 + A4 = 90.848
44 A5 = 90.848
45 A3 + A4 = 120.711
46 A5 = 120.711
47 A3 + A4 = 154.546
48 A5 = 154.546
49 A3 + A4 = 192.355
50 A5 = 192.355

```

```
51 A3 + A4 = 234.141
52 A5    = 234.141
```

Scopriamo quindi piacevolmente che il teorema di Pitagora vale non solo per i quadrati ma per ogni poligono regolare.

6.6 Prestazioni

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 steso 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.