

## 6 Seconda lezione

Ripetiamo tutti insieme: Python conta da zero.

### 6.1 Gli array

Un array unidimensionale è semplicemente una sequenza ordinata di numeri; è, in sostanza, un vettore e come tale si comporta. Utilizzeremo la libreria numpy. Per alcuni aspetti essi sono simili alle liste native di Python ma le differenze sono molte, in seguito ne vedremo alcune. Cominciamo con qualche esempio:

```
1 import numpy as np
2
3 #Creiamo un array di 5 elementi
4 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0]) #scrivere 2.0 equivale a scrivere 2.
5
6 print(array1)
7
8 #per accedere a un singolo elemento dell'array basta fare come segue:
9 elem = array1[1]
10
11 #ATTENZIONE! Gli indici, per Python, partono da 0, non da 1!
12 print(elem)
13
14 [Output]
15 [ 1.  2.  4.  8. 16.]
16 2.0
```

ora, avendo creato il nostro array potremmo volendo aggiungere o togliere degli elementi:

```
1 import numpy as np
2
3 array1=np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 #Aggiungiamo ora un numero in una certa posizione dell'array:
6 array1 = np.insert(array1, 4, 18)
7 '''
8 abbiamo aggiunto il numero 18 in quarta posizione, la sintassi e' :
9 np.insert(array a cui vogliamo aggiungere un numero, posizione dove aggiungerlo, numero)
10 '''
11 print(array1)
12
13 #Per aggiungere elementi in fondo ad un array esiste anche il comando append della libreria
    numpy:
14 array2 = np.append(array1, -4.)
15 print(array2)
16 #Mentre per togliere un elemento basta indicare il suo indice alla funzione remove di numpy:
17 array2 = np.delete(array2, 0)
18 print(array2)
19
20 [Output]
21 [ 1.  2.  4.  8. 18. 16.]
22 [ 1.  2.  4.  8. 18. 16. -4.]
23 [ 2.  4.  8. 18. 16. -4.]
```

Analogamente ciò può essere fatto per le liste con le funzioni native, quindi non di numpy, append e pop. Più corretto sarebbe dire che esse sono dei metodi della classe che gestisce le liste, infatti come vediamo nel prossimo esempio la sintassi è leggermente diversa, ma non vale la pena complicarci troppo la vita.

```
1 # lista iniziale
2 lista = [1, 2, 3, 4]
3 print(lista)
4
5 #aggiungo un elemento in coda, quindi alla fine della lista
6 lista.append(42)
7 print(lista)
8
9 #rimuovo l'ultimo elemento
10 lista.pop()
11 print(lista)
12
13 [Output]
14 [1, 2, 3, 4]
15 [1, 2, 3, 4, 42]
16 [1, 2, 3, 4]
```

Altri metodi interessanti per le liste sono "index()", "remove()", "count()", "insert()", "reverse()", "extend()", "sort()"; divertitevi a scoprire cosa ciascuno fa.

## 6.2 Tipi di array

Come le variabili numeriche sopra anche gli array posseggono i tipi e qui viene la prima differenza con le liste, se ad un array di numeri provassimo ad aggiungere un elemento che sia una stringa avremmo un errore; questo perché ogni array di numpy ha un suo tipo ben definito, che viene fissato, implicitamente o esplicitamente, al momento della creazione. Possiamo sì creare un array di tipo misto ma con tale array non si potrebbero fare le classiche operazioni matematiche.

```
1 import numpy as np
2
3 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 tipoarray1 = array1.dtype
6 print(tipoarray1)
7
8 a = np.array([0, 1, 2])
9 #abbiamo scritto solo numeri interi => array di interi
10
11 b = np.array([0., 1., 2.])
12 #abbiamo scritto solo numeri con la virgola => array di numeri float
13
14 """
15 #nota: anche se si dice "numero con la virgola",
16 vanno scritti sempre col punto!
17 La virgola separa gli argomenti
18 """
19
20 c = np.array([0, 3.14, 'giallo'])
21 #quest'array e' misto. Ci sono sia numeri interi che float che stringhe
22
23
24 #ora invece il tipo viene definito in maniera esplicita:
25 d = np.array([0., 1., 2.], 'int')
26 e = np.array([0, 1, 2], 'float')
27
28 print(a, a.dtype)
29 print(b, b.dtype)
30 print(c, c.dtype)
31 print(d, d.dtype)
32 print(e, e.dtype)
33
34
35 [Output]
36 float64
37 [0 1 2] int32
38 [0. 1. 2.] float64
39 ['0' '3.14' 'giallo'] <U32
40 [0 1 2] int32
41 [0. 1. 2.] float64
```

## 6.3 Array predefiniti

Vediamo brevemente alcuni tipi di array già definiti e di uso comune:

```
1 import numpy as np
2
3 #array contenente tutti zero
4 arraydizeri_0 = np.zeros(3)#il numero specificato e' la lunghezza
5 arraydizeri_1 = np.zeros(3, 'int')
6
7 #array contenente tutti uno
8 arraydiuni_0 = np.ones(5)#il numero specificato e' la lunghezza
9 arraydiuni_1 = np.ones(5, 'int')
10
11 print(arraydizeri_0, arraydizeri_1)
12 print(arraydiuni_0, arraydiuni_1)
13
14 """
15 questo invece e' un array il cui primo elemento e' zero
16 e l'ultimo elemento e' 1, lungo 10 e i cui elementi sono
17 equispaziati in maniera lineare tra i due estremi
18 """
19 equi_lin = np.linspace(0, 1, 10)
20 print(equi_lin)
21
```

```

22
23 """
24 questo invece e' un array il cui primo elemento e' 10^1
25 e l'ultimo elemento e' 10^2, lungo 10 e i cui elementi sono
26 equispaziati in maniera logaritmica tra i due estremi
27 """
28 equi_log = np.logspace(1, 2, 10)
29 print(equi_log)
30
31 [Output]
32 [0.  0.  0.] [0 0 0]
33 [1.  1.  1.  1.  1.] [1 1 1 1 1]
34 [0.          0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
35  0.66666667 0.77777778 0.88888889 1.          ]
36 [ 10.          12.91549665  16.68100537  21.5443469   27.82559402
37  35.93813664  46.41588834  59.94842503  77.42636827 100.          ]

```

## 6.4 Operazioni con gli array

Vediamo ora un po' di cose che si possono fare con gli array:

```

1 import numpy as np
2
3 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 primi_tre = array1[0:3]
6 print('primi_tre = ', primi_tre)
7 """
8 Questa sintassi seleziona gli elementi di array1
9 dall'indice 0 incluso all'indice 3 escluso.
10 Il risultato e' ancora un array.
11 """
12
13 esempio = array1[1:-1]
14 print(esempio)
15 esempio = array1[-2:5]
16 print(esempio)
17 #Questo metodo accetta anche valori negativi, con effetti curiosi
18
19
20 elementi_pari = array1[0::2]
21 print('elementi_pari = ', elementi_pari)
22 """
23 In questo esempio invece, usando invece due volte il simbolo :
24 intendiamo prendere solo gli elementi dall'indice 0 saltando di 2 in 2.
25 Il risultato e' un array dei soli elementi di indice pari
26 """
27
28 rewind = array1[::-1]
29 print('rewind = ', rewind)
30 """
31 Anche qui possiamo usare valori negativi.
32 In particolare questo ci permette di saltare "all'indietro"
33 e, ad esempio, di invertire l'ordine di un'array con un solo comando
34 """
35
36 [Output]
37 primi_tre = [1.  2.  4.]
38 [2.  4.  8.]
39 [ 8. 16.]
40 elementi_pari = [ 1.  4. 16.]
41 rewind = [16.  8.  4.  2.  1.]

```

Benché strano python non rifiuta gli indici negativi semplicemente perché essi indicizzano l'array al contrario ma partendo da 1. Quindi "-1" indica l'ultimo elemento, "-2" il penultimo e così via. Grande comodità sono le operazioni matematiche che possono essere fatte direttamente senza considerare i singoli valori, o meglio Python ci pensa da sé a fare le operazioni elemento per elemento. Gli array devono avere la stessa dimensione altrimenti avremmo errore, infatti potrebbe esserci un elemento spaato.

```

1 import math
2 import numpy as np
3
4 v = np.array([4, 5, 6])
5 w = np.array([1.2, 3.4, 5.8])
6

```

```

7 #classiche operazioni
8 somma = v + w
9 sottr = v - w
10 molt = v * w
11 div = v / w
12
13 print(v, w)
14 print()
15 print(somma, sottr, molt, div)
16 print()
17 #altri esempi
18 print(v**2)
19 print(np.log10(w))
20
21 """
22 come dicevamo prima qui' otterremmo errore poiche'
23 math lavora solo con numeri o, volendo,
24 array unidimensionali lunghi uno
25 """
26 print(math.log10(w))
27
28 [Output]
29 [4 5 6] [1.2 3.4 5.8]
30
31 [ 5.2  8.4 11.8] [2.8 1.6 0.2] [ 4.8 17.  34.8] [3.33333333 1.47058824 1.03448276]
32
33 [16 25 36]
34 [0.07918125 0.53147892 0.76342799]
35 Traceback (most recent call last):
36   File "<tmp 1>", line 26, in <module>
37     print(math.log10(w))
38 TypeError: only size-1 arrays can be converted to Python scalars

```

Se provassimo le stesse con delle liste solo la somma non darebbe errore, ma il risultato non sarebbe comunque lo stesso che otteniamo con gli array. Anche moltiplicare un array o una lista per un numero intero produce risultati diversi se provate vi sarà facile capire perché si è specificato che il numero deve essere intero. Ai fisici piace dire che un vettore è un qualcosa che trasforma come un vettore, e con questi esempi potrete capire che una lista non "trasforma" come un vettore mentre un array di numpy sì. Per questo nella programmazione scientifica se ne fa largo uso.

## 6.5 Maschere

Vi sarete di certo accorti che per creare un array di numpy quel che facciamo è passare una lista alla funzione "np.array" e infatti prima avevamo visto un array che conteneva una stringa, e per cui tutti gli elementi erano diventate stringhe. Similmente è possibile fare un array di valori booleani, cioè vero e falso.

```

1 import numpy as np
2
3 # array che contiene valori logici, detti booleani
4 b = np.array([True, False])
5
6 print(b)
7
8 [Output]
9 [ True False]

```

E fin qui nulla di sorprendente si potrebbe dire. Possiamo fare però una cosa interessante:

```

1 import numpy as np
2
3 # array che contiene valori logici, detti booleani
4 b = np.array([True, False])
5
6 # normalissimo array numerico
7 x = np.array([32, 89])
8 y = x[b] # x in corrispondenza di indici di b
9 print(y)
10
11 [Output]
12 [32]

```

Vediamo che se passiamo come indice del nostro array un array di valori logici, otteniamo un nuovo array, il quale contiene solo gli elementi corrispondenti all'indice che coincide con i valori "True" all'interno dell'array logico. Abbiamo creato quella che si dice una maschera e può essere utili in molti casi, vediamo un altro esempio.

```

1 import numpy as np
2
3 # creo un array
4 x = np.array([5, 4, 2, 8, 3, 9, 7, 2, 6, 3, 9, 8])
5
6 # voglio selezionare solo gli elementi maggiori di una certa soglia
7 mask = x >= 4 # mask e' un array di booleani, secondo la condizione data
8 # mask vale True negli indici in cui il valore di x e' maggiore o uguale a 4
9
10 print(x)
11 print(mask)
12 print(x[mask])
13
14 [Output]
15 [5 4 2 8 3 9 7 2 6 3 9 8]
16 [ True  True False  True False  True  True False  True False  True  True]
17 [5 4 8 9 7 6 9 8]

```

Poi se i nostri dati fossero una funzione del tempo potremmo magari creare la maschera sul tempo e applicarla ai nostri dati, magari perchè vogliamo considerare solo un certo range. Finché gli array son lunghi uguali potete fare quello che volete.

## 6.6 Matrici

Se un array unidimensionale lungo  $n$  è un vettore ad  $n$  componenti allora un array bidimensionale sarà una matrice.

```

1 import numpy as np
2
3 #esiste la funzione apposita di numpy per scrivere matrici.
4 matrice1 = np.matrix('1 2; 3 4; 5 6')
5 #Si scrivono essenzialmente i vettori riga della matrice separati da ;
6
7 #equivalente a:
8 matrice2 = np.matrix([[1, 2], [3, 4], [5,6]])
9 # oppure anche: matrice2 = np.array([[1, 2], [3, 4], [5,6]])
10 print(matrice1)
11 print(matrice2)
12
13
14 matricedizeri = np.zeros((3, 2)) #tre righe, due colonne: matrice 3x2
15 print('Matrice di zeri:\n', matricedizeri, '\n')
16 matricediuni = np.ones((3,2))
17 print('Matrice di uni:\n', matricediuni, '\n')
18
19 [Output]
20 [[1 2]
21  [3 4]
22  [5 6]]
23 [[1 2]
24  [3 4]
25  [5 6]]
26 Matrice di zeri:
27 [[0. 0.]
28  [0. 0.]
29  [0. 0.]
30
31 Matrice di uni:
32 [[1. 1.]
33  [1. 1.]
34  [1. 1.]

```

E ovviamente anche qui possiamo fare le varie operazioni matematiche:

```

1 import numpy as np
2
3 matrice1 = np.matrix('1 2; 3 4; 5 6')
4 matricediuni = np.ones((3,2))
5
6 sommadimatrici = matrice1 + matricediuni
7 print('Somma di matrici:\n', sommadimatrici)
8
9 matrice3 = np.matrix('3 4 5; 6 7 8') #matrice 2x3
10 prodottodimatrici = matrice1 * matrice3 #matrice 3x(2x2)x3
11 # attenzione che non funziona se si usa np.array()
12 #alternativamente si potrebbe scrivere: prodottodimatrici = matrice1 @ matrice3

```

```

13 # che funziona sia con np.matrix che np.array
14 print('\nProdotto di matrici:\n', prodottodimatrici)
15
16 [Output]
17 Somma di matrici:
18 [[2. 3.]
19  [4. 5.]
20  [6. 7.]]
21
22 Prodotto di matrici:
23 [[15 18 21]
24  [33 40 47]
25  [51 62 73]]

```

Ci siamo fermati alle matrici, oggetti a due indici, ma volendo avremmo potuto creare oggetti a più indici (i famosi tensori, tanto sempre numeri sono) ad esempio `np.ones((3,3,3))` creerebbe un oggetto a tre indici di uni, che possiamo vedere ad esempio come un vettore a tre componenti, ciascuna delle quali è una matrice  $3 \times 3$ . Se questo vi sembra strano aspettate di fare meccanica quantistica e ne riparlamo.

Ora è importante far notare una cosa: l'operazione di assegnazione con gli array (o liste) è delicata:

```

1 import numpy as np
2
3 a = np.array([1, 2, 3, 4])
4 print(f"array iniziale: {a}, id: {id(a)}")
5
6 b = a
7 b[0] = 7
8
9 print(f"array iniziale: {a}, id: {id(a)}")
10 print(f"array finale : {b}, id: {id(b)}")
11
12 #usiamo ora copy invece che l'assegnazione
13
14 a = np.array([1, 2, 3, 4])
15 print(f"array iniziale: {a}, id: {id(a)}")
16
17 b = np.copy(a)
18 b[0] = 7
19
20 print(f"array iniziale: {a}, id: {id(a)}")
21 print(f"array finale : {b}, id: {id(b)}")
22
23 [Output]
24 array iniziale: [1 2 3 4], id: 2226551695088
25 array iniziale: [7 2 3 4], id: 2226551695088
26 array finale : [7 2 3 4], id: 2226551695088
27 array iniziale: [1 2 3 4], id: 2226573280912
28 array iniziale: [1 2 3 4], id: 2226573280912
29 array finale : [7 2 3 4], id: 2226578310224

```

Come vedete se usiamo l'operato di assegnazione anche l'array iniziale cambia poiché sia a che b sono riferiti allo stesso indirizzo di memoria, mentre usando la funzione "copy" ora il secondo array ha un diverso indirizzo e quindi il problema non si pone più.

## 6.7 Esercizi

Ora che grazie agli array abbiamo un po' più di carne al fuoco voglio proporvi un paio di esercizi da fare da voi per prendere familiarità con questi oggetti, o strutture dati volendo. In basso ci sarà anche la soluzione che gradirei non guardaste prima di aver fatto almeno un tentativo. Ovviamente per fare questi esercizi non dovete usare informazioni ulteriori a quelle apprese fin'ora.

1. Verificare che anche con le liste l'operazione di assegnazione fa coincidere gli indirizzi di memoria.
2. 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)`.
3. Creare una matrice  $L_{4 \times 4}$  tutta nulla ma la cui diagonale sia: `[-1, 1, 1, 1]` e vedere cosa succede facendo  $v^T L v$ .
4. Data una matrice di zeri  $4 \times 2$  (4 righe, 2 colonne), riempite la colonna di sinistra e poi invertite le colonne.
5. Dato un `linspace` (e.g. tra 0 e 20) creare due nuovi array con solo i numeri pari uno e solo i dispari l'altro.
6. Dato un array a vostro piacimento, creare una maschera, con una o più condizioni sempre a vostro gusto.
7. Creare una matrice che contenga sulle colonne le tabelline da 0 a 10 (non banale da fare in poche righe).

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 I = np.zeros((3, 3))
4 idx = [0, 1, 2] # lista degli indici
5
6 I[idx, idx] = 1
7
8 v = np.array([4, 8, 2]) # vettore a caso
9 # calcolo prodotto scalare
10 print(v.T @ I @ v)
11 print(sum(v**2))
```

Terzo:

```
1 import numpy as np
2
3 L = np.zeros((4, 4))
4 idx = [0, 1, 2, 3] # lista degli indici
5
6 L[idx, idx] = [-1, 1, 1, 1]
7
8 v = np.array([4, 8, 2, 1]) # vettore a caso
9 print(v.T @ L @ v)
10 v = np.array([9, 0, 2, 1]) # vettore a caso
11 print(v.T @ L @ v)
```

Quarto:

```
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]])
```

Quinto:

```
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)
```

Sesto:

```
1 import numpy as np
2
3 x = np.linspace(0, 20, 21, dtype=int)
4
5 # maschera con due condizioni
6 mask = (x > 5) & (x < 15)
7 # "&" per and mentre "|" per or
8 print(x[mask])
```

Settimo:

```
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)
```