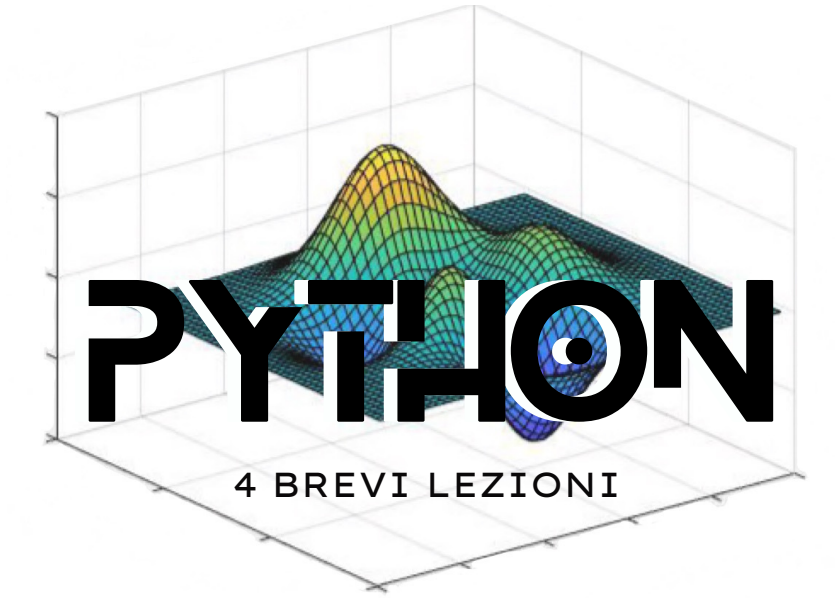
4 Brevi lezioni di Python

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 Seconda lezione

Ripetiamo tutti insieme: Python conta da zero.

1.1 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:

```
import numpy as np

#Creiamo un array di 5 elementi
array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0]) #scrivere 2.0 equivale a scrivere 2.

print(array1)

#per accedere a un singolo elemento dell'array basta fare come segue:
elem = array1[1]

#ATTENZIONE! Gli indici, per Python, partono da 0, non da 1!
print(elem)

[Output]
[ 1.  2.  4.  8. 16.]
2.0
```

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

```
import numpy as np

array1=np.array([1.0, 2.0, 4.0, 8.0, 16.0])

#Aggiungiamo ora un numero in una certa posizione dell'array:
array1 = np.insert(array1, 4, 18)
'''
abbiamo aggiunto il numero 18 in quarta posizione, la sintassi e' :
np.insert(array a cui vogliamo aggiungere un numero, posizione dove aggiungerlo, numero)
'''
print(array1)

#Per aggiungere elementi in fondo ad un array esiste anche il comando append della libreria
    numpy:
array2 = np.append(array1, -4.)
print(array2)
#Mentre per togliere un elemento basta indicare il suo indice alla funzione remove di numpy:
array2 = np.delete(array2, 0)
print(array2)

[Output]
[ 1.  2.  4.  8. 18. 16.]
[ 1.  2.  4.  8. 18. 16. -4.]
[ 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.

```
# lista iniziale
lista = [1, 2, 3, 4]
print(lista)

#aggiungo un elemento in coda, quinid alla fine della lista
lista.append(42)
print(lista)

#rimuovo l'ultimo elemento
lista.pop()
print(lista)

[Output]
[1, 2, 3, 4]
[1, 2, 3, 4, 42]
[1, 2, 3, 4]
```

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

1.2 Tipi di array

Come le variabile 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.

```
import numpy as np

array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])

tipoarray1 = array1.dtype
print(tipoarray1)

a = np.array([0, 1, 2])
#abbiamo scritto solo numeri interi => array di interi

b = np.array([0., 1., 2.])
#abbiamo scritto solo numeri con la virgola => array di numeri float

"""
#nota: anche se si dice "numero con la virgola",
vanno scritti sempre col punto!
La virgola separa gli argomenti
"""

c = np.array([0, 3.14, 'giallo'])
#quest'array e' misto. Ci sono sia numeri interi che float che stringhe


#ora invece il tipo viene definito in maniera esplicita:
d = np.array([0., 1., 2.], 'int')
e = np.array([0, 1, 2], 'float')

print(a, a.dtype)
print(b, b.dtype)
print(c, c.dtype)
print(d, d.dtype)
print(e, e.dtype)


[Output]
float64
[0 1 2] int32
[0. 1. 2.] float64
['0' '3.14' 'giallo'] <U32
[0 1 2] int32
[0. 1. 2.] float64
```

1.3 Array predefiniti

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

```
import numpy as np

#array contenente tutti zero
arraydizeri_0 = np.zeros(3)#il numero specificato e' la lunghezza
arraydizeri_1 = np.zeros(3, 'int')

#array contenente tutti uno
arraydiuni_0 = np.ones(5)#il numero specificato e' la lunghezza
arraydiuni_1 = np.ones(5, 'int')

print(arraydizeri_0, arraydizeri_1)
print(arraydiuni_0, arraydiuni_1)

"""
questo invece e' un array il cui primo elemento e' zero
e l'ultimo elemento e' 1, lungo 10 e i cui elementi sono
equispaziati in maniera lineare tra i due estremi
"""
equi_lin = np.linspace(0, 1, 10)
print(equi_lin)

```

```
"""
questo invece e' un array il cui primo elemento e' 10^1
e l'ultimo elemento e' 10^2, lungo 10 e i cui elementi sono
equispaziati in maniera logaritmica tra i due estremi
"""
equi_log = np.logspace(1, 2, 10)
print(equi_log)

[Output]
[0. 0. 0.] [0 0 0]
[1. 1. 1. 1. 1.] [1 1 1 1 1]
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
[ 10.          12.91549665  16.68100537  21.5443469   27.82559402
  35.93813664  46.41588834  59.94842503  77.42636827 100.        ]
```

1.4 Operazioni con gli array

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

```
import numpy as np

array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])

primi_tre = array1[0:3]
print('primi_tre = ', primi_tre)
"""
Questa sintassi seleziona gli elementi di array1
dall'indice 0 incluso all'indice 3 escluso.
Il risultato e' ancora un array.
"""

esempio = array1[1:-1]
print(esempio)
esempio = array1[-2:5]
print(esempio)
#Questo metodo accetta anche valori negativi, con effetti curiosi


elementi_pari = array1[0::2]
print('elementi_pari = ', elementi_pari)
"""
In questo esempio invece, usando invece due volte il simbolo :
intendiamo prendere solo gli elementi dall'indice 0 saltando di 2 in 2.
Il risultato e' un array dei soli elementi di indice pari
"""

rewind = array1[::-1]
print('rewind = ', rewind)
"""
Anche qui possiamo usare valori negativi.
In particolare questo ci permette di saltare "all'indietro"
e, ad esempio, di invertire l'ordine di un'array con un solo comando
"""

[Output]
primi_tre =  [1. 2. 4.]
[2. 4. 8.]
[ 8. 16.]
elementi_pari =  [ 1.  4. 16.]
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.
Per quelli che me l'hanno chiesto a lezione, il motivo per cui la scrittura `::-1` funziona per invertire l'ordine degli elementi, in generale, deriva dal fatto che l'operazione di slicing in generale prende 3 parametri, così indicati:

arr[start:stop:step]

ora, sappiamo bene che cosa fanno **`start`** e **`stop`** (definiscono, rispettivamente, l'indice di partenza incluso e l'indice di arrivo escluso), tuttavia **`step`** (che può anche non essere specificato e in tal caso viene preso uguale a 1) è curioso, siccome definisce la spaziatura fra un indice e l'altro fra gli elementi dell'array "tagliuzzato" fra **`start`** e stop. Per chiarire bene cosa voglia dire questa frase, consideriamo il seguente esempio

```
a = [2, 4, 6, 8, 10, 12, 14]
print(a[::2])

[Output]
[2, 6, 10, 14]
```

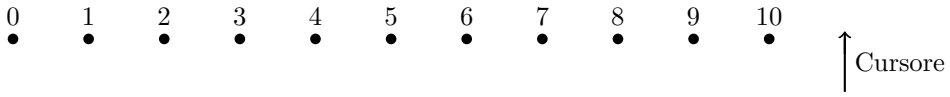
ovvero questo programma va a stampare a schermo muovendosi di 2 in 2 fra gli elementi di una lista: lo `step` definisce di quanto si sposta Python nell'array "tagliuzzato" fra un elemento e un altro, tenendo a mente che questa operazione di scorrere fra gli elementi di un array avviene **dopo** aver tagliuzzato l'array (cosa molto importante per capire come mai `::-1` funziona). Possiamo anche usare assieme `start`, `stop` e `step` come possiamo vedere in questo esempio

```
a = list(range(0, 100)) # convertito in lista cosi' stampava direttamente i numeri
print(a[30:80:2]) # stampa i numeri fra 30 (incluso) e 80 (escluso) di 2 in 2.
[Output]
[30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74,
    76, 78]
```

e, come possiamo intuire, può essere comodo in alcune situazioni. Tornando al fulcro della domanda, il nostro interprete prende l'intervallo di interi compreso fra `start` e `stop` scorrendo fra un elemento e un altro con un certo `step`: questa operazione restituirà una lista vuota se non esiste una "strada" per raggiungere lo `stop` con quel determinato `step` preso in questione. Questo che ho appena detto vi apparirà più chiaro con il seguente esempio

```
a = list(range(10)) # a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a[9:0:-1])
print(a[0:9:-1])
[Output]
[9, 8, 7, 6, 5, 4, 3, 2, 1] # a[9:0:-1]
[] # a[0:9:-1]
```

dove, nel caso del primo array, stiamo scorrendo fra 9 e 0 spostandosi di -1, conseguentemente scorrendo di -1 eventualmente raggiungeremo $stop = 0$. Nel secondo caso Python inizia a scorrere partendo da 0 ma non esiste un intero maggiore di 0 tale da fare in modo che $0 + n * (-1) \geq 9$, pertanto vi restituirà una lista vuota siccome non esiste una strada per farvi raggiungere il 9. Dunque la scrittura `::-1` funziona (e il disegnino qua sotto vi potrà forse chiarire le idee) siccome, se non specifichiamo *start* e *stop*, il nostro array viene "lasciato" immutato e, proprio per questo, mettendo uno `step` negativo il cursore "immaginario" che rappresenta lo scorrere dell'interprete lungo la lista verrà posizionato alla fine della lista (visto che l'interprete farà una delle sue magie mettendo uno `stop` compatibile con lo `step` non avendo specificato né `start` né `stop` per rendere possibile lo slice) e, dunque, inizierà ad iterare all'indietro.



a[11::-1] = a[::-1]

commento personale: non posso scrivere quello che ho già detto a lezione, ma ricompenserò con una lauta *sorpresa* la persona che mi mostrerà di saper far funzionare programmi di un certo livello con step negativi, siccome a parere mio è un gran casino (a livello logico e concettuale) scrivere programmi e risolvere dei bug legati a problemi con slicing a step negativi.
Grande comodità sono le operazione 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 spaiato.

```
import math
import numpy as np

v = np.array([4, 5, 6])
w = np.array([1.2, 3.4, 5.8])

#classiche operazioni
somma = v + w
sottr = v - w
molt = v * w
div = v / w

print(v, w)
print()
print(somma, sottr, molt, div)
print()
```

```
#altri esempi
print(v**2)
print(np.log10(w))

"""
come dicevamo prima qui' otterremmo errore poiche'
math lavora solo con numeri o, volendo,
array unidimensionali lunghi uno
"""
print(math.log10(w))

[Output]
[4 5 6] [1.2 3.4 5.8]

[ 5.2  8.4 11.8] [2.8 1.6 0.2] [ 4.8 17.  34.8] [3.33333333 1.47058824 1.03448276]

[16 25 36]
[0.07918125 0.53147892 0.76342799]
Traceback (most recent call last):
  File "<tmp 1>", line 26, in <module>
    print(math.log10(w))
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 capite che una lista non "trasforma" come un vettore mentre un array di numpy sì. Per questo nella programmazione scientifica se ne fa largo uso.

1.5 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.

```
import numpy as np

# array che contiene valori logici, detti booleani
b = np.array([True, False])

print(b)

[Output]
[ True False]
```

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

```
import numpy as np

# array che contiene valori logici, detti booleani
b = np.array([True, False])

# normalissimo array numerico
x = np.array([32, 89])
y = x[b] # x in corrispondenza di indici di b
print(y)

[Output]
[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 è può essere utili in molti casi, vediamo un altro esempio.

```
import numpy as np

# creo un array
x = np.array([5, 4, 2, 8, 3, 9, 7, 2, 6, 3, 9, 8])

# voglio selezionare solo gli elementi maggiori di una certa soglia
mask = x >= 4 # mask e' un array di booleani, secondo la condizione data
# mask vale True negli indici in cui il valore di x e' maggiore o uguale a 4

print(x)
```

```
print(mask)
print(x[mask])

[Output]
[5 4 2 8 3 9 7 2 6 3 9 8]
[ True  True False  True False  True  True False  True False  True  True]
[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.

1.6 Matrici

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

```
import numpy as np

#esiste la funzione apposita di numpy per scrivere matrici.
matrice1 = np.matrix('1 2; 3 4; 5 6')
#Si scrivono essenzialmente i vettori riga della matrice separati da ;

#equivalente a:
matrice2 = np.matrix([[1, 2], [3, 4], [5,6]])
# oppure anche: matrice2 = np.array([[1, 2], [3, 4], [5,6]])
print(matrice1)
print(matrice2)


matricedizeri = np.zeros((3, 2)) #tre righe, due colonne: matrice 3x2
print('Matrice di zeri:\n', matricedizeri, '\n')
matricediuni = np.ones((3,2))
print('Matrice di uni:\n', matricediuni, '\n')

[Output]
[[1 2]
 [3 4]
 [5 6]]
[[1 2]
 [3 4]
 [5 6]]
Matrice di zeri:
 [[0. 0.]
 [0. 0.]
 [0. 0.]]

Matrice di uni:
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
```

E ovviamente anche qui possiamo fare le varie operazioni matematiche:

```
import numpy as np

matrice1 = np.matrix('1 2; 3 4; 5 6')
matricediuni = np.ones((3,2))

sommadimatrici = matrice1 + matricediuni
print('Somma di matrici:\n', sommadimatrici)

matrice3 = np.matrix('3 4 5; 6 7 8') #matrice 2x3
prodottodimatrici = matrice1 * matrice3  #matrice 3x(2x2)x3
# attenzione che non funziona se si usa np.array()
#alternativamente si potrebbe scrivere: prodottodimatrici = matrice1 @ matrice3
# che funziona sia con np.matrix che np.array
print('\nProdotto di matrici:\n', prodottodimatrici)

[Output]
Somma di matrici:
 [[2. 3.]
 [4. 5.]
 [6. 7.]]

Prodotto di matrici:
```

```
 [[15 18 21]
 [33 40 47]
 [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 3×3. Se questo vi sembra strano aspettate di fare meccanica quantistica e ne riparliamo.
Ora è importante far notare una cosa: l'operazione di assegnazione con gli array (o liste) è delicata:

```python
import numpy as np

a = np.array([1, 2, 3, 4])
print(f"array iniziale: {a}, id: {id(a)}")

b = a
b[0] = 7

print(f"array iniziale: {a}, id: {id(a)}")
print(f"array finale  : {b}, id: {id(b)}")

#usiamo ora copy invece che l'assegnazione

a = np.array([1, 2, 3, 4])
print(f"array iniziale: {a}, id: {id(a)}")

b = np.copy(a)
b[0] = 7

print(f"array iniziale: {a}, id: {id(a)}")
print(f"array finale  : {b}, id: {id(b)}")

[Output]
array iniziale: [1 2 3 4], id: 2226551695088
array iniziale: [7 2 3 4], id: 2226551695088
array finale  : [7 2 3 4], id: 2226551695088
array iniziale: [1 2 3 4], id: 2226573280912
array iniziale: [1 2 3 4], id: 2226573280912
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.

1.7 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.

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

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

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 fra 0 e 20 inclusi) 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.

See you Space Cowboy ...