5 Seconda lezione

Ripetiamo tutti insieme: Python conta da zero.

5.1 Gli array

Un array unidimensionale è semplicemente una sequenza ordinata di numeri; è, possiamo dire, un vettore. 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.]
```

5.2 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
```

5.3 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.        ]
```

5.4 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.]
```

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.

5.5 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]])

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
#alternativamente si potrebbe scrivere: prodottodimatrici = matrice1 @ matrice3

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]]
```

Ora è importante far notare una cosa: l'operazione di assegnazione con gli array è delicata, anche con le liste:

```
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 iniziale: {a}, 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] = 1674

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

[Output]
array iniziale: [1 2 3 4], id: 1377943201616
array iniziale: [7 2 3 4], id: 1377943201616
array iniziale: [7 2 3 4], id: 1377943201616
array iniziale: [1 2 3 4], id: 1377948097296
array iniziale: [1 2 3 4], id: 1377948097296
array iniziale: [1 2 3 4], id: 1377948097872
```

Come vedete se usiamo l'operato di assegnazione anche l'array iniziale cambia poiche 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.