INDICE

Numpy e' un modulo non-standard (nel senso che non e' distribuito insieme all'interprete) ma di fatto ha assunto il ruolo di principale modulo per l'esecuzione di analisi numeriche.
Nel seguito importeremo il modulo con un alias e ci riferiremo ad esso con `np`

In [2]:
```
import numpy as np
```

Alcuni link utili:

- Documentazione ufficiale numpy
- Lista di esempi catalogati per funzione
- Thesaurus of Mathematical Languages or MATLAB synonymous commands in Python/NumPy

La classe ndarray

L'elemento fondamentale del modulo Numpy e' la classe `ndarray` che ha le seguenti caratteristiche:

- contiene dati di tipo omogeneo
- il numero degli elementi deve essere noto nel momento della creazione (anche se puo' essere modificato con non indifferenti costi computazionali)
- puo' essere multidimensionale
- gli elementi contenuti nell'array si indicizzano con numeri interi e slices e, come vedremo anche con altri tipi di oggetto
- supportano la cosiddetta *vettorizzazione* che, attraverso l'eliminazione della necessita' di iterare in Python sugli elementi dell'array, puo' incrementare notevolmente la velocita' di esecuzione

Il dtype

Ad ogni array è associato un oggetto che descrive come sono organizzati e come devono essere interpretati i byte nel blocco di memoria corrispondente a ciascun elemento dell' array. Questo descrittore è detto dtype.

Il modulo numpy rende disponibili vari dtypes predefiniti che permettono di controllare le dimensioni in byte di ciascun elemento.

Data type	Description
bool	Boolean (True or False) stored as a byte
int	Platform integer (normally either `int32` or `int64`)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float	Shorthand for `float64`.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex	Shorthand for `complex128`.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

Come vedremo in seguito è possibile creare un dtypes che rappresenti oggetti più complicati (ad esempio istanze di una classe definita dall'utente).

primi passi e principali attributi

creo un array da una lista con la funzione 'array'

In [3]:
```
#creo un array a partire da una lista
a = np.array( [1,2,3] )
```

In [4]:
```
a
```

Out[4]:
```
array([1, 2, 3])
```

l'array ha molti attributi:

In [6]:
```
#dir(a)
```

iniziamo ad elencarne i principali:

- ndim (intero): il numero di dimensioni (o meglio il numero di 'axis') dell'array (1 per un vettore, 2 per una matrice, ...)
- shape (toupla di ndim interi): le dimensioni dell array (le dimensioni di ciascuno degli 'axis')
- size (intero): il numero totale di elementi dell'array
- dtype (oggetto dtype): un oggetto che indica il tipo di dato degli elementi dell'array.
- itemsize (intero): le dimensioni in byte di ciascun elemento
- data (oggetto memoryview): un riferimento alla porzione di memoria che contiene i dati

In [7]:
```
print('a =',a)
print('a.ndim ({0}) = {1}'.format(type(a.ndim),a.ndim))
print('a.shape ({0}) = {1}'.format(type(a.shape),a.shape))
print('a.size ({0}) = {1}'.format(type(a.size),a.size))
print('a.dtype ({0}) = {1}'.format(type(a.dtype),a.dtype))
print('a.itemsize ({0}) = {1}'.format(type(a.itemsize),a.itemsize))
print('a.data ({0}) = {1}'.format(type(a.data),a.data))
```

```
a = [1 2 3]
a.ndim (<class 'int'>) = 1
a.shape (<class 'tuple'>) = (3,)
a.size (<class 'int'>) = 3
a.dtype (<class 'numpy.dtype'>) = int32
a.itemsize (<class 'int'>) = 4
a.data (<class 'memoryview'>) = <memory at 0x02D68080>
```

In [8]:
```
a = np.array([ [1,2,3] , [4,5,6] ])
print('a =',a)
print('a.ndim ({0}) = {1}'.format(type(a.ndim),a.ndim))
print('a.shape ({0}) = {1}'.format(type(a.shape),a.shape))
print('a.size ({0}) = {1}'.format(type(a.size),a.size))
print('a.dtype ({0}) = {1}'.format(type(a.dtype),a.dtype))
print('a.itemsize ({0}) = {1}'.format(type(a.itemsize),a.itemsize))
print('a.data ({0}) = {1}'.format(type(a.data),a.data))
```

```
a = [[1 2 3]
 [4 5 6]]
a.ndim (<class 'int'>) = 2
a.shape (<class 'tuple'>) = (2, 3)
a.size (<class 'int'>) = 6
a.dtype (<class 'numpy.dtype'>) = int32
a.itemsize (<class 'int'>) = 4
a.data (<class 'memoryview'>) = <memory at 0x02D69030>
```

vediamo gli stessi attributi per un array bidimensionale

In [9]:
```
a = np.array([ [1,2,3] , [4,5,6] ]) # matrice 2x3
print('a.ndim ({0}) = {1}'.format(type(a.ndim),a.ndim))
print('a.shape ({0}) = {1}'.format(type(a.shape),a.shape))
print('a.size ({0}) = {1}'.format(type(a.size),a.size))
print('a.dtype ({0}) = {1}'.format(type(a.dtype),a.dtype))
print('a.itemsize ({0}) = {1}'.format(type(a.itemsize),a.itemsize))
print('a.data ({0}) = {1}'.format(type(a.data),a.data))
```

```
a.ndim (<class 'int'>) = 2
a.shape (<class 'tuple'>) = (2, 3)
a.size (<class 'int'>) = 6
a.dtype (<class 'numpy.dtype'>) = int32
a.itemsize (<class 'int'>) = 4
a.data (<class 'memoryview'>) = <memory at 0x02D69300>
```

creare un ndarray

Per un elenco completo delle routines usate per la creazione di array vedi il seguente link routines per la creazione di array

Ci sono almeno quattro possibili strategie per la creazione di array:

- Conversione da altre strutture Python compatibili (ad esempio, liste, tuple)
- Funzioni specifiche NumPy (ad esempio, arange, ones, zeros , ecc)
- Lettura array dal disco (da formati standard e non)
- Creazione di array di byte non elaborati attraverso l'uso di stringhe o buffer

1) Conversione da altre strutture Python compatibili: la funzione array

la funzione array accetta diversi parametri di cui uno solo (object) e' strettamente necessario:

numpy.array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)

uso di base

passo una lista di interi

In [10]:
```
a=np.array([1,2,3])#NB le parentesi quadre sono necessarie!!
print(a)
print(a.dtype)
```

```
[1 2 3]
int32
```

passo una lista di interi e float ...

In [11]:
```
a=np.array([1,2,3.])
print(a)
print(a.dtype)
```

```
[ 1.  2.  3.]
float64
```

specifico il dtype

posso esplicitare il tipo

In [12]:
```
a=np.array([1,2,3],dtype=float64)
print(a)
print(a.dtype)
```

```
[ 1.  2.  3.]
float64
```

... ma non tutte le conversioni sono permesse:

In [13]:
```
a=np.array( [ 1 , 2 , 3+0j ] , dtype=int32)
print(a)
print(a.dtype)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-e770f80acc41> in <module>()
----> 1 a=np.array( [ 1 , 2 , 3+0j ] , dtype=int32)
      2 print(a)
      3 print(a.dtype)

TypeError: can't convert complex to int
```

copia

se passo ad array un nd array la funzione restituisce una copia ...

In [15]:
```
b=np.array(a)


print(b is a)
print('-'*15)
print(a)
print(b)
b[0]=100
print('-'*15)
print(a)
print(b)
```

```
False
---------------
[ 1.  2.  3.]
[ 1.  2.  3.]
---------------
[ 1.  2.  3.]
[ 100.    2.    3.]
```

... a meno che gli dica di non farlo

In [16]:
```
b=np.array(a,copy=False)
b is a

print(b is a)
print('-'*15)
print(a)
print(b)
```

```
b[0]=100
print('-'*15)
print(a)
print(b)
```

```
True
---------------
[ 1.  2.  3.]
[ 1.  2.  3.]
---------------
[ 100.    2.    3.]
[ 100.    2.    3.]
```

ma se la copia e' necessaria (ad esempio un cambio di tipo) viene comunque effettuata

In [17]:
```
b=np.array(a,copy=False,dtype=complex)
b is a
```

Out[17]:
```
False
```

1) bis uso i dtypes numpy (vedi tabella sopra) come funzioni

possiamo usare i dtype predefiniti come se fossero funzioni ...

funziona sia con numeri singoli

In [18]:
```
a=1
b=np.int64(a)
type(a),type(b)
```

Out[18]:
```
(builtins.int, numpy.int64)
```

sia con liste o altre sequenze

In [19]:
```
a=[1,2,3]
b=np.float64(a)
print(a,type(a),type(a[0]))
print(b,type(b),type(b[0]))
```

```
[1, 2, 3] <class 'list'> <class 'int'>
[ 1.  2.  3.] <class 'numpy.ndarray'> <class 'numpy.float64'>
```

con liste annidate

In [20]:
```
a=[[1,2,3],[4,5,6]]
b=np.float64(a)
print(a,type(a),type(a[0]))
print(b,type(b),type(b[0]))
```

```
[[1, 2, 3], [4, 5, 6]] <class 'list'> <class 'list'>
[[ 1.  2.  3.]
 [ 4.  5.  6.]] <class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

i due statement seguenti sono equivalenti:

In [ ]:
```
b = np.float64(a)
c = np.array(a,dtype=float64)
```

2) funzioni specifiche NumPy (ad esempio, arange, ones, zeros , ecc)

sequenze

http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html#numerical-ranges

numpy.arange

uso base (ha la stessa sintassi della funzione builtin range):

In [21]:
```
a = np.arange(10)
print(a)
a.dtype
```

```
[0 1 2 3 4 5 6 7 8 9]
```

Out[21]:
```
dtype('int32')
```

con start e step e dtype esplicito

In [22]:
```
a = np.arange(1,10,2,dtype=int8)
print(a)
a.dtype
```

```
[1 3 5 7 9]
```

Out[22]:
```
dtype('int8')
```

con float

In [23]:
```
a = arange(1.3 ,10.34 ,0.2 )
print(a)
print(a.dtype)
```

```
[  1.3   1.5   1.7   1.9   2.1   2.3   2.5   2.7   2.9   3.1   3.3   3.5
   3.7   3.9   4.1   4.3   4.5   4.7   4.9   5.1   5.3   5.5   5.7   5.9
   6.1   6.3   6.5   6.7   6.9   7.1   7.3   7.5   7.7   7.9   8.1   8.3
   8.5   8.7   8.9   9.1   9.3   9.5   9.7   9.9  10.1  10.3]
float64
```

numpy.linspace(start, stop, num=50, endpoint=True, retstep=False)

punti equispaziati tra due estremi

In [24]:
```
np.linspace(2.0, 3.0, num=5) # cinque punti compreso stop
```

Out[24]:
```
array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])
```

di default anche il punto finale stop è compreso ma ...

In [25]:
```
np.linspace(2.0, 3.0, num=5, endpoint=False) # cinque punti escluso stop
```

Out[25]:
```
array([ 2. ,  2.2,  2.4,  2.6,  2.8])
```

volendo restituisce anche lo step tra due elementi contigui:

In [26]:
```
np.linspace(2.0, 3.0, num=5, retstep=True)
```

Out[26]:
```
(array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

uni e zeri

http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html#ones-and-zeros

numpy.ones(shape, dtype=None, order='C')

crea array di soli 1

vettore di cinque elementi (il dtype predefinito è float)

In [27]:
```
np.ones(5)
```

Out[27]:
```
array([ 1.,  1.,  1.,  1.,  1.])
```

matrice di 5x4x3 elementi interi

In [146]:
```
np.ones((5,3,2),dtype=np.int8)
```

Out[146]:
```
array([[[1, 1],
        [1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1],
```

```
       [1, 1]]], dtype=int8)
```

numpy.ones_like(a, dtype=None, order='K', subok=True)¶

crea un array uni con forma uguale a quella di un altro array

In [30]:
```
a = np.array( [ [1,2,3] , [4,5,6] ] )
np.ones_like(a)
```

Out[30]:
```
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

cambio tipo

In [31]:
```
a = np.array( [ [1,2,3] , [4,5,6] ] )
np.ones_like(a,dtype=complex)
```

Out[31]:
```
array([[ 1.+0.j,  1.+0.j,  1.+0.j],
       [ 1.+0.j,  1.+0.j,  1.+0.j]])
```

vedi anche zeros, zeros_like, empty, empty_like, identity, eye

numpy.fromfunction(function, shape, **kwargs)

La funzione fromfunction crea un array con forma specificata e li riempie con il valore restituito da una funzione che accetta come parametri gli indici che individuano ciascun elemento dell'array.

array monodimensionale

In [34]:
```
def myfunc1(i):
    """restituisce
       i**2 se i>=5
       i    altrimenti"""
    print(type(i))
    print('i=',i)
    return (i>=5)*(i**2)+(i<5)*(i)

a = np.fromfunction(function = myfunc1 , shape=(10,) , dtype=int32)

print('a=',a)
```

```
<class 'numpy.ndarray'>
i= [0 1 2 3 4 5 6 7 8 9]
a= [ 0  1  2  3  4 25 36 49 64 81]
```

array bidimensionale

In [35]:
```
def myfunc2(i,j):
    print('i =\n', i)
    print('j =\n', j)
    return i==j

a = np.fromfunction(myfunc2,(5,5))
print('a=\n',a)
```

```
i =
 [[ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]
 [ 2.  2.  2.  2.  2.]
 [ 3.  3.  3.  3.  3.]
 [ 4.  4.  4.  4.  4.]]
j =
 [[ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]]
a=
 [[ True False False False False]
 [False  True False False False]
 [False False  True False False]
 [False False False  True False]
 [False False False False  True]]
```

Gli esempi sopra li capiremo meglio una volta che avremo parlato delle operazioni base sugli array. Soprattutto capiremo come le funzioni passate a fromfunction non lavorano su indici ma su array di indici. Ciò può essere evidenziato rimuovendo i commenti.

3) Lettura array dal disco (da formati standard e non)

Quelle sotto sono le principali funzioni di base (GENERICHE) disponibili in numpy.

- numpy.ndarray.fromfile
- numpy.loadtxt
- numpy.genfromtxt

Per salvare su file:

- numpy.ndarray.tofile
- numpy.savetxt

esempio di tofile/fromfile

Prima di leggere un array da un file dobbiamo prima scrivere il file stesso tofile (ATTENZIONE ALLE NOTE)

In [36]:
```
a=np.array([[1,2,3],[4,5,6]],dtype=np.int8)
print(a.shape)
a.tofile('a.bin')
```

```
(2, 3)
```

lo leggo con fromfile:

In [37]:
```
a2 = np.fromfile('a.bin',dtype=np.int8)
```

In [38]:
```
a2
```

Out[38]:
```
array([1, 2, 3, 4, 5, 6], dtype=int8)
```

... non mantiene la shape ...

esempio di loadtxt

genero un file di testo adatto

In [39]:
```
np.savetxt("a.txt", a)
```

In [40]:
```
a2 = genfromtxt("a.txt")
a2
```

Out[40]:
```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

Per una panoramica sulle tecniche di lettura/scrittura degli array su/da files vedi Cookbook InputOutput

4) Creazione di array di byte non elaborati attraverso l'uso di stringhe o buffer

Quelle sotto sono le principali funzioni di base (GENERICHE) disponibili in numpy.

- numpy.fromstring
- numpy.frombuffer nota: i buffer sono oggetti che permettono l'accesso ai dati grezzi in memoria (bytres e bytesarray ad esempio)

Indicizzazione e slicing

array monodimensionali

indicizzazione di base

l'indicizzazione di base avviene come per le altre sequenze di python gia' viste (liste e touple):

In [41]:
```
a=np.arange(10)
print(a)
print(a[2])
```

```
[0 1 2 3 4 5 6 7 8 9]
2
```

slicing

lo slicing avviene con le stesse modalita' viste per le liste

[start : end : stop]

In [44]:
```
a = np.arange(20)
print(a)
type(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Out[44]: numpy.ndarray

In [43]:
```
a[3:15:3] # NB:15 non incluso
```

Out[43]: array([3, 6, 9, 12])

valori di default per gli slice

se non specificati i tre indici che definiscono lo slice assumono i rispettivi valori di default, che sono, rispettivamente:

- start = 0
- stop = end (l'elemento successivo alla fine dell'array)
- step = 1

In [45]:
```
a=arange(20)
a[::3] #dall'inizio alla fine a passi di 3
```

Out[45]: array([0, 3, 6, 9, 12, 15, 18])

array multidimensionali

indicizzazione di base

Facciamo un esempio. Creo un array 2D

In [46]:
```
a = np.array([[1,2,3],[4,5,6]])
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

Per accedere ai suoi elementi uso le parentesi quadre e gli indici delle diverse dimensioni separati da virgole:

In [48]:
```
print(a[1,1])#seconda riga, seconda colonna
```

```
5
```

NOTA: nel caso delle liste annidate invece si usavano doppie parentesi graffe.

In [50]:
```
l=[[1,2,3],[4,5,6],[7,8,9]]
print(l)
print(l[1][1])
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
5
```

numero di indici minore del numero di dimensioni

per le liste annidate:

In [51]:
```
l[1]
```

Out[51]: [4, 5, 6]

per ndarray multidimensionali, se specifico un numero di indici minore delle dimensioni accade ciò che accade con le liste:

in 2 dimensioni:

In [52]:
```
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

In [53]:
```
print(a[1])
```

```
[4 5 6]
```

in 3 dimensioni:

In [54]:
```
a=np.array([ [[1,2,3],[4,5,6],[7,8,9]],
             [[11,22,33],[44,55,66],[77,88,99]],
             [[111,222,333],[444,555,666],[777,888,999]]])
a.shape
```

Out[54]: (3, 3, 3)

In [55]:
```
a[1]
```

Out[55]:
```
array([[11, 22, 33],
       [44, 55, 66],
       [77, 88, 99]])
```

In [56]:
```
a[1,1]
```

Out[56]: array([44, 55, 66])

slicing

in due dimensioni

In [57]:
```
#...esistono modi più efficienti x creare questo array ...
a=np.array([range(6),range(10,16),range(20,26),range(30,36),range(40,46),range(50,56)])
a
```

Out[57]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

estraggo la terza riga

In [58]:
```
a[2,:]#equivalente a a[2]
```

Out[58]: array([20, 21, 22, 23, 24, 25])

estraggo la terza colonna

In [60]:
```
a[:,2].reshape(6,1)
```

Out[60]:
```
array([[ 2],
       [12],
       [22],
       [32],
       [42],
       [52]])
```

cosa estraggo con lo slice seguente?

In [61]:
```
a
```

Out[61]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

In [62]:
```
a[-2: , -2:]
```

Out[62]:
```
array([[44, 45],
       [54, 55]])
```

In [ ]:
```
a
```

e cosi?

In [63]:
```
a[2:5:2 , :]
```

Out[63]:
```
array([[20, 21, 22, 23, 24, 25],
       [40, 41, 42, 43, 44, 45]])
```

e con questo?

In [65]:
```
a
```

Out[65]:
```
array([[ 0,  1,  2,  3,  4,  5],
```

```
               [10, 11, 12, 13, 14, 15],
               [20, 21, 22, 23, 24, 25],
               [30, 31, 32, 33, 34, 35],
               [40, 41, 42, 43, 44, 45],
               [50, 51, 52, 53, 54, 55]])
```

In [64]:
```
a[1:4:2 , ::2]
```

Out[64]:
```
array([[10, 12, 14],
       [30, 32, 34]])
```

in tre dimensioni

vedremo modi più efficienti per creare matrici del genere ma per adesso facciamo così:

In [69]:
```
a=np.array([
[range(6),range(10,16),range(20,26),range(30,36),range(40,46),range(50,56),range(60,66),range(70,76)],
[range(100,106),range(110,116),range(120,126),range(130,136),range(140,146),range(150,156),range(160,166),range(170,176

[range(200,206),range(210,216),range(220,226),range(230,236),range(240,246),range(250,256),range(260,266),range(270,276

])
#a = np.arange(6) + np.arange(0,80,10)[:,None] + np.arange(0,300,100)[:,None,None]
a
```

Out[69]:
```
array([[[  0,   1,   2,   3,   4,   5],
        [ 10,  11,  12,  13,  14,  15],
        [ 20,  21,  22,  23,  24,  25],
        [ 30,  31,  32,  33,  34,  35],
        [ 40,  41,  42,  43,  44,  45],
        [ 50,  51,  52,  53,  54,  55],
        [ 60,  61,  62,  63,  64,  65],
        [ 70,  71,  72,  73,  74,  75]],

       [[100, 101, 102, 103, 104, 105],
        [110, 111, 112, 113, 114, 115],
        [120, 121, 122, 123, 124, 125],
        [130, 131, 132, 133, 134, 135],
        [140, 141, 142, 143, 144, 145],
        [150, 151, 152, 153, 154, 155],
        [160, 161, 162, 163, 164, 165],
        [170, 171, 172, 173, 174, 175]],

       [[200, 201, 202, 203, 204, 205],
        [210, 211, 212, 213, 214, 215],
        [220, 221, 222, 223, 224, 225],
        [230, 231, 232, 233, 234, 235],
        [240, 241, 242, 243, 244, 245],
        [250, 251, 252, 253, 254, 255],
        [260, 261, 262, 263, 264, 265],
        [270, 271, 272, 273, 274, 275]]])
```

cosa estraggo?

In [67]:
```
a[1:2 , 3:5 , 2:4]
```

Out[67]:
```
array([[[132, 133],
        [142, 143]]])
```

e con questo?

In [70]:
```
a[: , ::2 , 1::2]
```

Out[70]:
```
array([[[  1,   3,   5],
        [ 21,  23,  25],
        [ 41,  43,  45],
        [ 61,  63,  65]],

       [[101, 103, 105],
        [121, 123, 125],
        [141, 143, 145],
        [161, 163, 165]],

       [[201, 203, 205],
        [221, 223, 225],
        [241, 243, 245],
        [261, 263, 265]]])
```

Stampa degli array

Consideriamo un esempio che permette di creare un array tridimensionale che contiene gli inici associati a ciascun elemento dell'array stesso:

In [71]:
```
nx=4
ny=5
nz=6
i = np.indices((nx,ny,nz))
mi=np.rec.fromarrays([i[0],i[1],i[2]],names='ix,iy,iz')
print('\n\nmi=')
print(mi)
```

```
mi=
[[[(0, 0, 0) (0, 0, 1) (0, 0, 2) (0, 0, 3) (0, 0, 4) (0, 0, 5)]
  [(0, 1, 0) (0, 1, 1) (0, 1, 2) (0, 1, 3) (0, 1, 4) (0, 1, 5)]
  [(0, 2, 0) (0, 2, 1) (0, 2, 2) (0, 2, 3) (0, 2, 4) (0, 2, 5)]
  [(0, 3, 0) (0, 3, 1) (0, 3, 2) (0, 3, 3) (0, 3, 4) (0, 3, 5)]
  [(0, 4, 0) (0, 4, 1) (0, 4, 2) (0, 4, 3) (0, 4, 4) (0, 4, 5)]]

 [[(1, 0, 0) (1, 0, 1) (1, 0, 2) (1, 0, 3) (1, 0, 4) (1, 0, 5)]
  [(1, 1, 0) (1, 1, 1) (1, 1, 2) (1, 1, 3) (1, 1, 4) (1, 1, 5)]
  [(1, 2, 0) (1, 2, 1) (1, 2, 2) (1, 2, 3) (1, 2, 4) (1, 2, 5)]
  [(1, 3, 0) (1, 3, 1) (1, 3, 2) (1, 3, 3) (1, 3, 4) (1, 3, 5)]
  [(1, 4, 0) (1, 4, 1) (1, 4, 2) (1, 4, 3) (1, 4, 4) (1, 4, 5)]]

 [[(2, 0, 0) (2, 0, 1) (2, 0, 2) (2, 0, 3) (2, 0, 4) (2, 0, 5)]
  [(2, 1, 0) (2, 1, 1) (2, 1, 2) (2, 1, 3) (2, 1, 4) (2, 1, 5)]
  [(2, 2, 0) (2, 2, 1) (2, 2, 2) (2, 2, 3) (2, 2, 4) (2, 2, 5)]
  [(2, 3, 0) (2, 3, 1) (2, 3, 2) (2, 3, 3) (2, 3, 4) (2, 3, 5)]
  [(2, 4, 0) (2, 4, 1) (2, 4, 2) (2, 4, 3) (2, 4, 4) (2, 4, 5)]]

 [[(3, 0, 0) (3, 0, 1) (3, 0, 2) (3, 0, 3) (3, 0, 4) (3, 0, 5)]
  [(3, 1, 0) (3, 1, 1) (3, 1, 2) (3, 1, 3) (3, 1, 4) (3, 1, 5)]
  [(3, 2, 0) (3, 2, 1) (3, 2, 2) (3, 2, 3) (3, 2, 4) (3, 2, 5)]
  [(3, 3, 0) (3, 3, 1) (3, 3, 2) (3, 3, 3) (3, 3, 4) (3, 3, 5)]
  [(3, 4, 0) (3, 4, 1) (3, 4, 2) (3, 4, 3) (3, 4, 4) (3, 4, 5)]]]
```

infatti:

In [72]:
```
mi[2,3,4]
```

Out[72]: (2, 3, 4)

Dall' esempio illustrato sopra si può notare che gli array sono stampati in modo che:

- l'ultimo indice a destra è stampato in modo da aumentare da sinistra a destra
- il penultimo indice a destra aumenta dall'alto verso il basso.
- gli indici ancora più a sinistra dall'alto verso il basso dove ciacun 'piano' è separato da una riga vuota.

inoltre

- L'ultimo indice a destra è quello che varia più velocemente
- L'ultimo indice a sinistra è quello che varia più lentamente

Viste e copie

Quando si opera sugli array a volte si ottengono delle copie a volte no. Priviamo ad approfondire la questione.

reference a array

In [73]:
```
a = np.array([[1,2,3],[4,5,6]])
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

vi ricorda qualcosa?

In [74]:
```
b = a
a[1]=[40,50,60]
print('a=',a)
print('b=',b)
```

```
a= [[ 1  2  3]
 [40 50 60]]
b= [[ 1  2  3]
 [40 50 60]]
```

passaggio di un array a funzioni (per riferimento):

In [75]:
```
def myfunc(m):
    m[0,0]=10

#la funzione modifica il contenuto dell'array
myfunc(a)
#le modifiche sono visibili all'esterno della funzione
print(a)
```

```
[[10  2  3]
 [40 50 60]]
```

viste di array

diversi ARRAY possono condividere ***gli stessi*** DATI

view

In tali casi si parla di diverse viste degli stessi dati.

In [76]:
```
a=np.array([[1,2,3],[4,5,6]])
a
```

Out[76]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

creo una vista con view()

In [77]:
```
b = a.view()
```

b ed a sono oggetti diversi:

In [78]:
```
b is a
```

Out[78]:
```
False
```

ma condividono gli stessi dati, infatti:

se cambio la forma di b

In [79]:
```
b.shape=1,6
b
```

Out[79]:
```
array([[1, 2, 3, 4, 5, 6]])
```

la forma di a non cambia

In [80]:
```
a
```

Out[80]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

se cambio il contenuto di b

In [81]:
```
b[0,0]=129
b
```

Out[81]:
```
array([[129,   2,   3,   4,   5,   6]])
```

In [82]:
```
a
```

Out[82]:
```
array([[129,   2,   3],
       [  4,   5,   6]])
```

flag.owndata e attributo base

il flag owndata ci dice se un array è una vista di un altro array (owndata == False) oppure no ...

In [83]:
```
a.flags.owndata , b.flags.owndata
```

Out[83]:
```
(True, False)
```

l'attributo base di una vista è un riferimento all'array 'osservato:

```
In [84]: b.base is a
```

```
Out[84]:  True
```

```
In [85]: a.base , b.base
```

```
Out[85]:  (None, array([[129,   2,   3],
                [  4,   5,   6]]))
```

vedi anche numpy.ndarray.base per il significato dell'attributo base

slices

A DIFFERENZA CHE PER LE LISTE, anche gli slice sono viste

```
In [97]: c = a[:,1]#la prima colonna
```

```
In [98]: c.base is a
```

```
Out[98]:  True
```

```
In [99]: c[1]=114
```

```
In [100]:a
```

```
Out[100]: array([[129,   2,   3],
                 [  4, 114,   6]])
```

deep copy

per fare una vera copia si può usare sia il metodo copy

```
In [90]: #copio
         d=a.copy()
         #modifico
         d[1,1]=99
         #stampo
         a,d
```

```
Out[90]:  (array([[ 129,    2,    3],
                  [   4, 1234,    6]]),
           array([[129,   2,   3],
                  [  4,  99,   6]]))
```

sia (come già detto) la funzione array

```
 In [ ]: #copio
         e=np.array(a)
         #modifico
         e[1,1]=999
         #stampo
         a,e
```

Manipolazione della shape

cambio di forma generico

reshape

restituisce una vista

il numero di elementi deve rimanere invariato

```
In [106]:a0 = np.arange(25)
          print(a0)
          a=a0.reshape(5,5)
          a[3,2]=100000
          print(a)
          print(a0)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
[[     0      1      2      3      4]
 [     5      6      7      8      9]
 [    10     11     12     13     14]
 [    15     16 100000     18     19]
 [    20     21     22     23     24]]
[     0      1      2      3      4      5      6      7      8      9
     10     11     12     13     14     15     16 100000     18     19
     20     21     22     23     24]
```

NB è una vista!!

In [ ]:
```
a[0,0]=100
a0
```

metodo resize

cambia la forma in place

In [2]:
```
a=arange(10)
print(type(a.resize(5,2)))
a
```

```
<class 'NoneType'>
```

Out[2]:
```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

aggiunge zeri se ci sono più elementi:

In [3]:
```
a=np.arange(10)
a.resize(5,5)
print(a)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
```

taglia se necessario

In [4]:
```
a.resize(5,1)
a
```

Out[4]:
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

funzione resize

accetta un array ed una shape come parametri e restituisce una copia!!!

In [5]:
```
b=np.resize(a,(10))
b[0]=10
print(b)
print(a)
```

```
[10  1  2  3  4  0  1  2  3  4]
[[0]
 [1]
 [2]
 [3]
 [4]]
```

ripete i dati se necessario:

In [6]:
```
np.resize(b,(7,3))
```

Out[6]:
```
array([[10,  1,  2],
       [ 3,  4,  0],
       [ 1,  2,  3],
       [ 4, 10,  1],
       [ 2,  3,  4],
       [ 0,  1,  2],
```

```
       [ 3,  4, 10]])
```

appiattire

flatten

In [7]:
```
b = a.flatten() #restituisce sempre una copia
b
```

Out[7]:
```
array([0, 1, 2, 3, 4])
```

ravel

In [8]:
```
c = a.ravel() #restituisce una vista, equivalente a a.reshape(-1)
```

In [9]:
```
c
```

Out[9]:
```
array([0, 1, 2, 3, 4])
```

iterare sugli elementi di un array appiattito

creo un array

In [119]:
```
a=arange(5*3*2).reshape(2,3,5).transpose()
a
```

Out[119]:
```
array([[[ 0, 15],
        [ 5, 20],
        [10, 25]],

       [[ 1, 16],
        [ 6, 21],
        [11, 26]],

       [[ 2, 17],
        [ 7, 22],
        [12, 27]],

       [[ 3, 18],
        [ 8, 23],
        [13, 28]],

       [[ 4, 19],
        [ 9, 24],
        [14, 29]]])
```

itera sugli elementi in sequenza sull'array appiattito nell'ordine standard (che è quello con cui gli elementi sono stampati)

In [120]:
```
for el in a.flat:
    print(el)
```

```
0
15
5
20
10
25
1
16
6
21
11
26
2
17
7
22
12
27
3
18
8
23
13
28
4
19
```

```
9
24
14
29
```

resize/reshape con ordine Fortran-like

Di default le operazioni di reshape/flattening avvengono usando un ordinamento 'C-like', secondo il quale l'indice che varia più velocemente è quello più a destra.

In [121]:
```
a=np.arange(15).reshape((5,3),order='C')
a
```

Out[121]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

torno indietro ...

In [122]:
```
fa=a.flatten(order='C')
fa
```

Out[122]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Tuttavia posso anche utilizzare un ordinamento dell'array 'FORTRAN-like':

In [123]:
```
a=np.arange(15).reshape((5,3),order='F')
a
```

Out[123]:
```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

torno indietro ...

In [124]:
```
fa=a.flatten(order='F')
fa
```

Out[124]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [125]:
```
a=np.arange(25).reshape((5,5),order='C')
a
```

Out[125]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

In [126]:
```
a=np.arange(25).reshape((5,5),order='F')
a
```

Out[126]:
```
array([[ 0,  5, 10, 15, 20],
       [ 1,  6, 11, 16, 21],
       [ 2,  7, 12, 17, 22],
       [ 3,  8, 13, 18, 23],
       [ 4,  9, 14, 19, 24]])
```

reshape con indicazione incompleta delle dimensioni

se specifico -1 in operazioni di reshaping per una delle dimensioni, questa viene determinata in base al numero totale degli elementi dell'array e a quello di tutte le altre dimensioni

In [127]:
```
#30 elementi organizzati in (3,10)
np.arange(30).reshape(3,-1)
```

Out[127]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])
```

come sopra

In [131]:
```
np.arange(30).reshape(10,-1)
```

Out[131]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23],
       [24, 25, 26],
       [27, 28, 29]])
```

Unire array

http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html#joining-arrays

unire array orizzontalmente (per colonne)

uso np.vstack

- gli array di partenza possono avere un diverso numero di elementi
- la funzione accetta come argomento una sequenza di array
- gli array da unire devono avere le stesse dimensioni tranne che per la seconda dimensione da sinistra (le colonne in 2D)

In [132]:
```
a=np.arange(10).reshape(5,-1)#5 righe, 2 colonne
b=np.zeros((5,6))# 5 righe, 6 colonne
print('a=',a)
print('a.shape=',a.shape)
print('b=',b)
print('b.shape=',b.shape)
```

```
a= [[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
a.shape= (5, 2)
b= [[ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
b.shape= (5, 6)
```

In [133]:
```
c=np.hstack((a,b,a))#5 righe 10 colonne
c
```

Out[133]:
```
array([[ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.],
       [ 2.,  3.,  0.,  0.,  0.,  0.,  0.,  0.,  2.,  3.],
       [ 4.,  5.,  0.,  0.,  0.,  0.,  0.,  0.,  4.,  5.],
       [ 6.,  7.,  0.,  0.,  0.,  0.,  0.,  0.,  6.,  7.],
       [ 8.,  9.,  0.,  0.,  0.,  0.,  0.,  0.,  8.,  9.]])
```

NB solo la seconda dimensione può contenere un numero diverso di elementi

In [134]:
```
a=np.ones((2,3,3))
b=np.zeros((2,1,3))
print('a=',a)
print()
print('b=',b)
```

```
a= [[[ 1.  1.  1.]
  [ 1.  1.  1.]
  [ 1.  1.  1.]]

 [[ 1.  1.  1.]
  [ 1.  1.  1.]
  [ 1.  1.  1.]]]

b= [[[ 0.  0.  0.]]

 [[ 0.  0.  0.]]]
```

In [136]:
```
c=np.hstack((a,b,a))
c.shape
```

Out[136]: (2, 7, 3)

unire array verticalmente (per righe)

uso np.vstack

- gli array di partenza possono avere un diverso numero di elementi
- la funzione accetta come argomento una sequenza di array
- gli array da unire devono avere le stesse dimensioni tranne che per la prima a sinistra nella shape

In [ ]:
```
a=np.zeros((1,5))#1 riga, 5 colonne
b=np.ones((2,5))#2 righe, 5 colonne
print('a=',a)
print('a.shape=',a.shape)
print('b=',b)
print('b.shape=',b.shape)
```

In [ ]:
```
c=np.vstack((a,b,a))#4 righe, 5 colonne
c
```

esempio in 3D (impilo per piani):

In [ ]:
```
a=np.zeros((1,5,3))
b=np.ones((2,5,3))
print('a=',a)
print('a.shape=',a.shape)
print('b=',b)
print('b.shape=',b.shape)
c=np.vstack((a,b,a))
c
print('c=',c)
print('c.shape=',c.shape)
```

Unire array per un generico asse

con concatenate posso unire secondo un generico asse

esempio: come vstack

In [138]:
```
a=np.zeros((1,5))#1 riga, 5 colonne
b=np.ones((2,5))#2 righe, 5 colonne
print('a=',a)
print('a.shape=',a.shape)
print()
print('b=',b)
print('b.shape=',b.shape)
```

```
a= [[ 0.  0.  0.  0.  0.]]
a.shape= (1, 5)

b= [[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]]
b.shape= (2, 5)
```

In [139]:
```
c=np.concatenate((a,b,a),0)

print('c=',c)
print('c.shape=',c.shape)
```

```
c= [[ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.]]
c.shape= (4, 5)
```

esempio: come h stack

In [140]:
```
a=np.arange(10).reshape(5,-1)#5 righe, 2 colonne
b=np.zeros((5,6))# 5 righe, 6 colonne
c=np.concatenate((a,b,a),1)#5 righe 10 colonne
c
```

Out[140]:
```
array([[ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.],
       [ 2.,  3.,  0.,  0.,  0.,  0.,  0.,  0.,  2.,  3.],
       [ 4.,  5.,  0.,  0.,  0.,  0.,  0.,  0.,  4.,  5.],
       [ 6.,  7.,  0.,  0.,  0.,  0.,  0.,  0.,  6.,  7.],
       [ 8.,  9.,  0.,  0.,  0.,  0.,  0.,  0.,  8.,  9.]])
```

esempio: unisco per il terzo asse

In [ ]:
```
a=np.ones((2,3,1))
b=np.zeros((2,3,2))
print('a=',a)
print('a.shape=',a.shape)
print()
print('b=',b)
print('b.shape=',b.shape)
```

In [ ]:
```
c=np.concatenate((a,b,a),2)
c
```

Dividere array

vedi:

http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html#splitting-arrays

Come nel caso dell'unione di array anche per la suddivisione ci sono delle funzioni che operano per righe, per colonne e per un generico asse. Vediamo solo quest'ultima

dividere un array 2D per righe

in parti uguali se possibile

In [141]:
```
a=zeros((8,4))
print('a=',a)
#divido in 2ue parti secondo l'asse 0
array_split(a,2,0)
```

```
a= [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

Out[141]:
```
[array([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]]),
 array([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])]
```

divido in in punti specifici

In [142]:
```
#divido in in punti specifici, secondo l'asse 0
array_split(a,[2,7],0)#divido prima del terzo e prima dell'ottavo elemento)
```

Out[142]:
```
[array([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]]),
 array([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]]),
 array([[ 0.,  0.,  0.,  0.]])]
```

dividere un array 3D

In [ ]:
```
a=arange(3*4*5).reshape(3,4,5)
a
```

per piani in 3 parti

In [ ]:
```
array_split(a,3,0)
```

per righe in 3 parti

In [ ]:
```
a
```

In [ ]:
```
array_split(a,3,1)
```

per colonne in 3 parti

In [ ]:
```
a
```

In [ ]:
```
array_split(a,3,2)
```

Operazioni di base

Operazioni tra array delle stesse dimensioni

avvengono elemento per elemento

somma

In [143]:
```
a=np.arange(5)
b=np.arange(10,15)
print('a  =',a)
print('b  =',b)
print('a+b=',a+b)
```

```
a  = [0 1 2 3 4]
b  = [10 11 12 13 14]
a+b= [10 12 14 16 18]
```

moltiplicazione

anche la moltiplicazione:

In [144]:
```
print('a*b=',a*b)
```

```
a+b= [ 0 11 24 39 56]
```

Ovviamente esiste anche la possibilità di effettuare il prodotto riga per colonna ma si deve fare riferimento a funzioni specifiche:

In [ ]:
```
np.dot(a,b)
```

etc etc

Operazioni con costanti

In [145]:
```
a=np.arange(25).reshape(5,5)
4*a
```

Out[145]:
```
array([[ 0,  4,  8, 12, 16],
       [20, 24, 28, 32, 36],
       [40, 44, 48, 52, 56],
       [60, 64, 68, 72, 76],
       [80, 84, 88, 92, 96]], dtype=int32)
```

In [ ]:
```
a+10
```