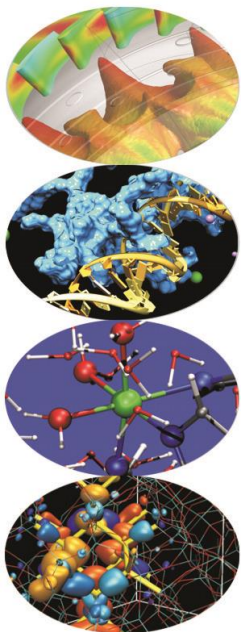
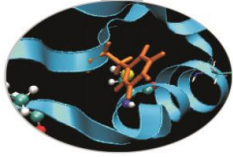




Python per il calcolo scientifico



Indice



- [Numpy](#)
- [Creazione di un array](#)
- [Reshape & Resize](#)
- [Indexing](#)
- [Operatori Aritmetici](#)
- [ufunc](#)
- [Broadcasting](#)
- [Cenni di vettorizzazione](#)
- [Efficienza e loop](#)
- [Array Copy](#)
- [Attributi e Metodi](#)
- [I/O con Array Numpy](#)
- [Matrix](#)



Python in ambito scientifico

Python è diventato accessibile a nuovi gruppi di utilizzatori. A dispetto della sua semplicità è un linguaggio abbastanza potente da permettere la gestione di applicazioni complesse, supportando la programmazione ad oggetti, la programmazione funzionale e la programmazione generica.

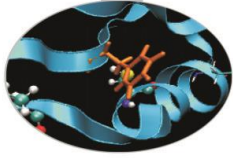
Oltre alla standard library sono stati sviluppati moduli ad hoc per il calcolo scientifico (Numpy, Pylab, Scipy...).

Rispetto a Matlab è un linguaggio general purpose, è completamente open source ed è altamente portabile.

A livello mondiale è sfruttato in ambito scientifico (e non solo) in diversi progetti:

- National Space Telescope Laboratory (Hubble Space Telescope)
- Lawrence Livermore National Laboratories (pyMPI)
- Enthought Corporation (Applicazioni Geofisiche ed Elettromagnetiche)
- Anaconda (RedHat Linux Installer)
- Google

Contenuti



Nello specifico verranno trattati:

- Introduzione base al modulo *numpy*: definizione di array e operazioni base.

Scaricabile:

<https://sourceforge.net/projects/numpy/files/>

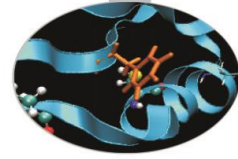
Official web-site:

<http://numpy.scipy.org/>

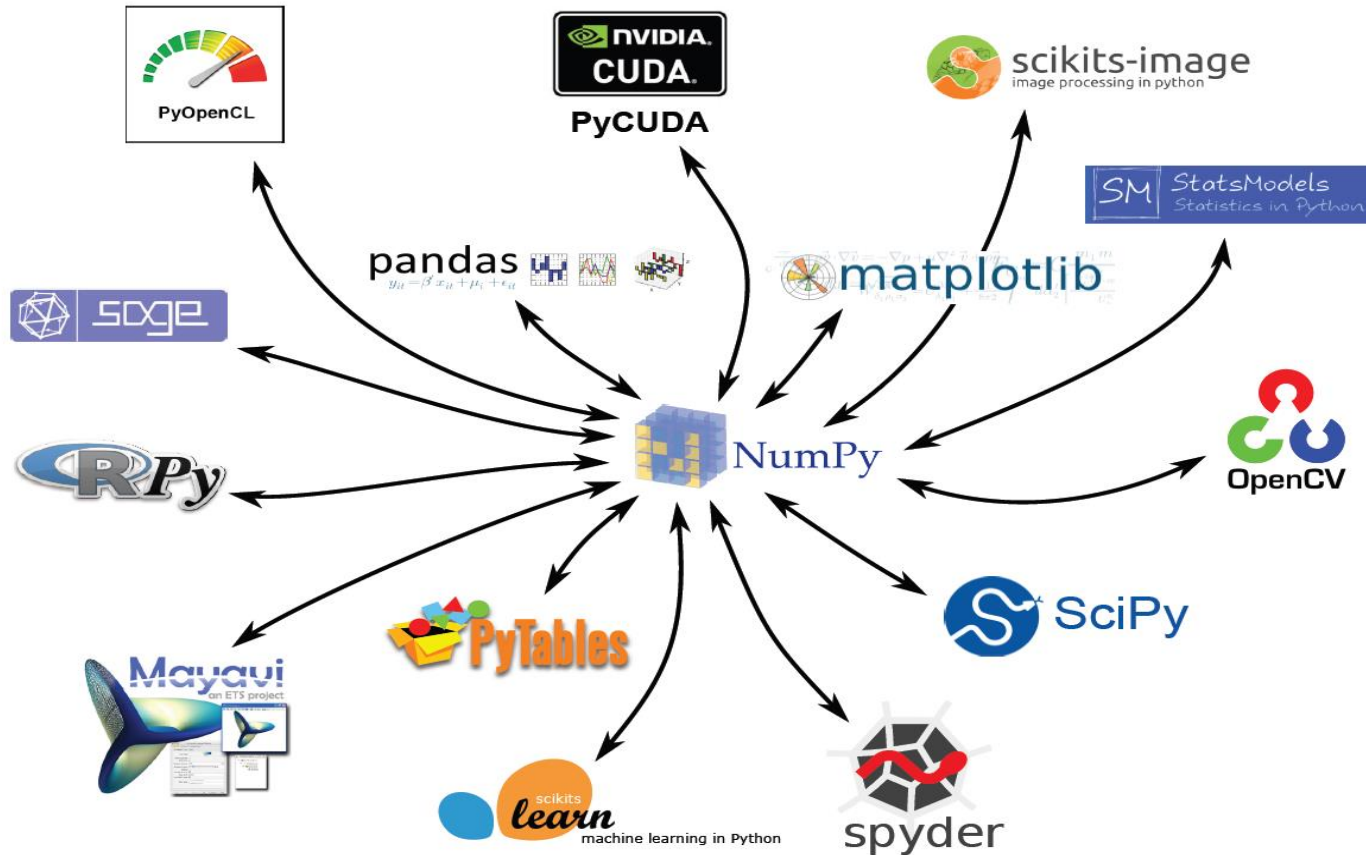
- Introduzione al modulo *pylab*: creazione di grafici

Official web-site:

<http://matplotlib.sourceforge.net/>



Numpy



“Life is too short to write C++ code”

David Beazley - EuroScipy 2012 Bruxelles

NumPy



NumPy è l'abbreviazione di *Numerical Python*: un'estensione del linguaggio pensata per l'ottimizzazione della gestione di grosse moli di dati, utilizzata principalmente in ambito scientifico.

Il modulo NumPy ingloba le features dei moduli Numeric e Numarray e aggiunge nuove funzionalità.

In questa overview verranno discusse solo le strutture dati **array**.

Il modulo *NumPy* fornisce un nuovo contenitore dati, particolarmente performante, la struttura *array*.

L'object *array* è una struttura dati omogenea (fixed-type) e multidimensionale.

Terminologia:

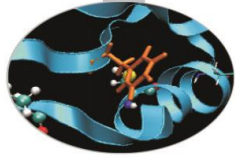
Con *size* di un array intendiamo il numero di elementi presenti in un array.

Con *rank* di un array si intende il numero di assi/dimensioni di un array.

Con *shape* di un array intendiamo le dimensioni dell'array, cioè una tupla di interi contenente il numero di elementi per ogni dimensione, numero VARIABILE.

L' *itemsize* rappresenta la dimensione in memoria di ogni singolo elemento dell'array.

Perché usare Numpy

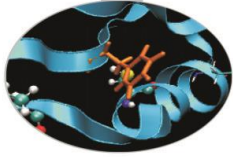


- In NumPy funzioni e metodi agiscono su interi vettori e matrici, quindi raramente si ricorre a loop espliciti, tipicamente poco efficienti
- Gli algoritmi utilizzati sono estremamente testati e disegnati con l'obiettivo di fornire alte prestazioni
- NumPy gestisce lo store degli array in memoria in modo molto più efficiente, rispetto a quanto accade in Python puro per le liste e le liste di liste
- Le operazioni di I/O di array sono significativamente più veloci
- File di grandi dimensioni possono essere *memory-mapped*, consentendo così una lettura/scrittura ottimale di grosse moli di dati
- Molte parti di NumPy sono scritte in C, cosa che rende un codice NumPy più veloce dell'analogo in Python puro
- NumPy prevede una C API, che consente di estenderne le funzionalità (questo tema non viene trattato in questa introduzione a NumPy)



Quando non usare Numpy

- Al di fuori del contesto del calcolo tecnico scientifico, l'uso di NumPy è meno utile
- Principali limitazioni:
 - NumPy non è supportato per lo sviluppo di applicazioni Google App Engine, perché molte sue parti sono scritte in C
 - Gli utenti di Jython – l'implementazione Java di Python – non possono contare sul modulo NumPy: Jython gira all'interno di una Java Virtual Machine e non può importare NumPy, perché molte sue parti sono scritte in C



Import del modulo

- Importiamo il modulo come di consueto

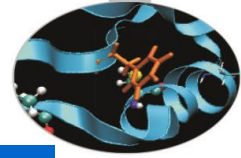
```
>>> import numpy
```

```
>>> from numpy import *
```

```
>>> import numpy as np    #default in molti codici e nella documentazione  
                           numpy
```

Numpy fornisce un nuovo tipo di dato: un array N-dimensionale (ndarray).

Organizzazione del modulo



Sub-Packages	Purpose	Comments
core	basic objects	all names exported to numpy
lib	Additional utilities	all names exported to numpy
linalg	Basic linear algebra	LinearAlgebra derived from Numeric
fft	Discrete Fourier transforms	FFT derived from Numeric
random	Random number generators	RandomArray derived from Numeric
distutils	Enhanced build and distribution	improvements built on standard distutils
testing	unit-testing	utility functions useful for testing
f2py	Automatic wrapping of Fortran code	a useful utility needed by SciPy

ndarray



La definizione del ndarray `numpy/core/include/numpy/ndarrayobject.h`

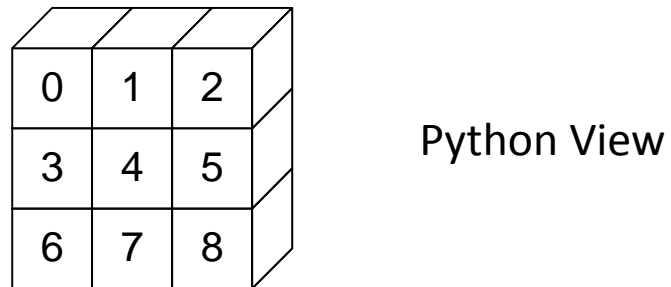
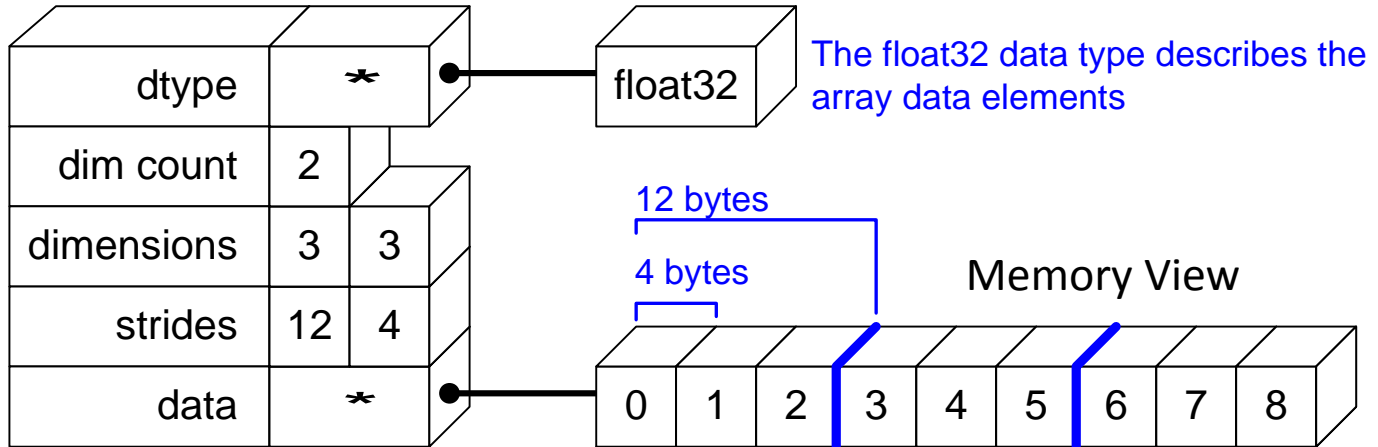
```
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;           // pointer to raw data buffer
    int nd;               // number of dimensions, also called ndim
    npy_intp *dimensions; // size in each dimension
    npy_intp *strides;    // bytes to jump to get to the next
                        // element in each dimension

    PyObject *base;

    PyArray_Descr *descr; /* Pointer to type structure */
    int flags;           /* Flags describing array */
    PyObject *weakreflist; /* For weakreferences */
} PyArrayObject;
```



ndarray





Creazione di un Array

Creazione di un array

Ci sono diverse modalità per generare un array. La più semplice consiste nell'utilizzo della funzione

`array(object, dtype=None, copy=1, order=None) → array`

NOTA: allocazione in memoria per array multidimensionali.

1	2	3
4	5	6

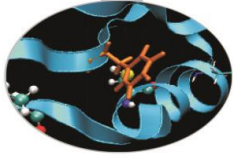
1	4	2	5	3	6
---	---	---	---	---	---

Fortran - Style

1	2	3	4	5	6
---	---	---	---	---	---

C-Style

Creazione di un Array



- Il modo più semplice per creare un array è di convertire altre strutture dati Python

```
>>>import numpy as np
```

```
>>>a=np.array([1,2,3,4])
```

```
>>>lista1=[1,2,3,4]
```

```
>>>tupla=(5,6,7,8)
```

```
>>>a=np.array(lista)    #from a list
```

```
>>>b=np.array(tupla)    #from a tupla
```

```
>>>c=np.array([lista,tupla]) #from a list and from a tupla
```

```
>>>a.dtype
```

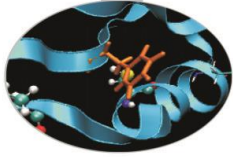
```
dtype('int32')
```



ndtype

- Ci sono *21 built-in data type* che possono essere usati per creare un array.
- *Numpy* supporta più tipi di dato rispetto a python puro.

Type	Description
bool	Boolean
int	Platforma integer
int8	Byte (-128,127)
int16	Integer (-32768,32767)
int32	Integer (-2147483648, 2147483647)
int64	Integer (-9223372036854775808, 9223372036854775807)
uint8	unsigned integer (0,255)
uint16	unsigned integer (0,65535)
uint32	unsigned integer (0,4294967295)
uint64	unsigned integer (0,18446744073709551615)
float	float64
float32	Single precision float
float64	Double precision float
complex	complex128
complex64	Complessi con 2 32-bits float
complex128	Complessi con 2 64-bits float



ndtype

- E' possibile creare array con nuovi dtype definiti dall'utente.

```
>>>dt=dtype([('Name','S3'),('Anni', numpy.int64)])
```

```
>>a=array([('Chiara',3),('Marco',4)],dtype=dt)
```

```
>>> a
```

```
array([('Chi', 3L), ('Mar', 4L)],
```

```
      dtype=[('Name', '|S3'), ('Val', '<i8')])
```

```
>>>dt=dtype({'names':('Name','Anni','Cognome'),'formats':('S3','int64','S3')})
```

```
>>>a=array([('Chiara',3,'Bianchi'),('Marco',4,'Rossi']),dtype=dt)
```

```
>>> a
```

```
array([('Chiara', 3L, 'Bia'), ('Marco', 4L, 'Ros')],
```

```
      dtype=[('Name', '|S10'), ('Anni', '<i8'), ('Cognome', '|S10')])
```


Dimension e shape



```
int nd;          /* number of dimensions, also called ndim */  
numpy_intp *dimensions; /* size in each dimension */
```

- L'attributo *shape* specifica la forma dell'array:

```
import numpy as np  
a=np.array([[1,2],[2,2]])  
a.shape  
(2,2)  
b=array([[[1,2],[3,4]],[[5,6],[7,8]]])  
b.shape
```

- L'attributo *ndim* specifica la dimensione dell'array

```
a.ndim  
2  
b.ndim  
3
```



itemsize

L'itemsize permette di specificare la dimensione di ciascun elemento

```
>>>b=array([[1, 2,3],[3, 4,5]])
```

```
>>> b.itemsize
```

```
4
```

```
>>> b.dtype
```

```
dtype('int32')
```

```
>>> b.strides
```

#bytes to jump to get to the next element
#of each dimension

```
(12, 4)
```

#skyp_byte_row, skype_byte_col

```
>>>c=array([[1,2,3],[4,5,6]],order='Fortran')
```

```
>>> c.strides
```

```
(4, 8)
```

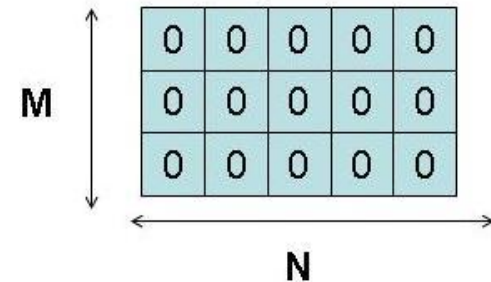


Creazione di un Array

Se il contenuto di un array è a priori ignoto, è utile utilizzare funzioni per riempire in modo sistematico l'array.

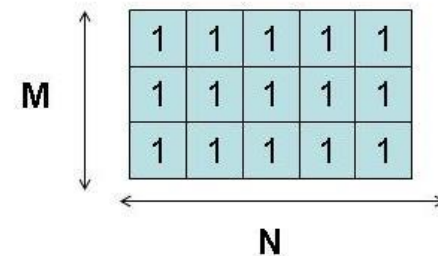
- La funzione *zeros* crea un array di dimensioni *shape* di soli zeri.

zeros(shape, dtype=float, order='C')



- La funzione *ones* crea un array di dimensione *shape* di soli uni.

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

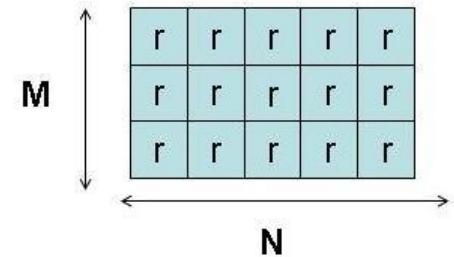




Creazione di un Array

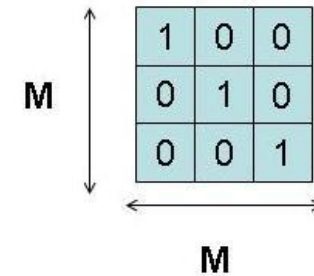
- La funzione *empty* crea un array di dimensioni *shape* senza inizializzazione.

empty(shape, dtype=None, order='C')



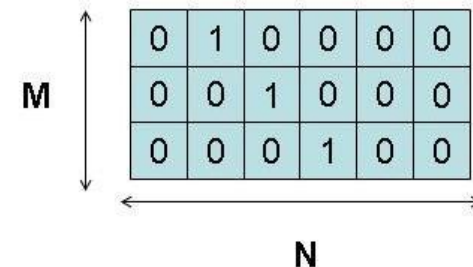
- La funzione *identity* genera la matrice identità $n \times n$

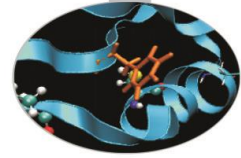
identity(n, dtype=None)



- La funzione *eye* crea una matrice $N \times M$ riempiendo di 1 la k -esima diagonale

eye(N, M=None, k=0, dtype=float)





Creazione di un Array

In maniera analoga alla funzione *range* per le liste, esiste la possibilità di creare una sequenza di numeri anche per gli array.

arange([start,] stop[, step,], dtype=None)

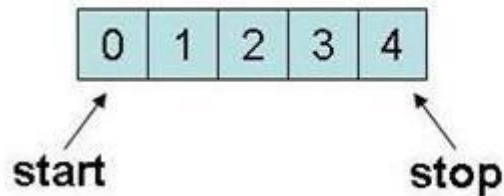
- La funzione *arange* genera un array di numeri compresi tra *start* e *stop* con passo *step*.

In alternativa:

linspace(start, stop, num=50, endpoint=True, restep=False)

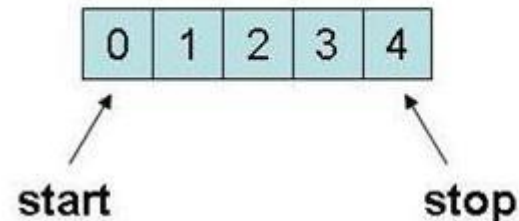
- Genera una sequenza di *num* numeri uniformemente distribuita compresa tra *start* e *stop*.

arange(0,5,1)



Step=1

linspace(0,4,5)



Num=5

Creazione di un Array



- E' possibile creare array anche da stringhe con la funzione fromstring

```
>>> np.fromstring('1 2', dtype=int, sep=' ')  
array([1, 2])
```

```
>>> np.fromstring('1, 2', dtype=int, sep=',')  
array([1, 2])
```

- E' possibile creare array leggendo i dati da disco...



Reshape & Resize

Reshaping & Resizing Array

I metodi `resize` e `reshape` permettono di modificare la forma e la dimensione dell'array. Il metodo

`reshape(shape, order='C')`

Restituisce una nuova struttura dati ridistribuendo gli elementi dell'array secondo la nuova forma *shape* con ordine *order*.

La funzione `reshape` deve lasciare invariato il numero di elementi dell'array.

La funzione

`resize(new_shape, refcheck=True, order=False)`

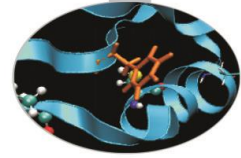
Lavora direttamente *in-place* e permette di modificare la forma dell'array e di ridimensionarlo.

`Resize` funziona solo se l'array non è referenza o non è referenziato.

Esempio

```
>>>a=arange(20)
```

```
>>>a.resize(5,6)          #Ok
```



Reshape & Resize

```
>>>b=a
```

```
>>>a.resize(3,3) #Error a is referenced by b
```

Traceback (most recent call last):

```
File "<pyshell#160>", line 1, in <module>
```

```
a.resize(3,3)
```

ValueError: cannot resize an array that has been referenced or is referencing another array in this way. Use the resize function

Esempio

```
>>>a=array(range(1,9))
```

```
>>>print "Shape" , a.shape
```

```
>>>c_style=a.reshape((2,2,2),order='C')
```

#Array Method: Numpy Style

```
>>>f_style=reshape(a,(2,2,2),order='F')
```

#Numpy Function

```
>>>print "C-style" , c_style
```

```
>>>print "Fortran-style" , f_style
```

```
>>>c_style=c_style.reshape((2,4))
```

```
>>>print "Reshape c_style", c_style
```

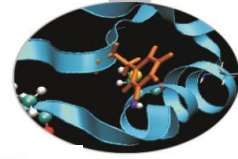
```
>>>f_style=f_style.reshape((2,4))
```

```
>>>print "Reshape f_style",f_style
```



array(range(1,9))

Reshape & Resize



OUTPUT

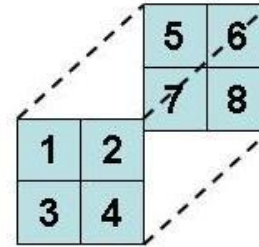
Shape (8,)

C-style `c_style [[[1, 2],
[3, 4],
[5, 6],
[7, 8]]]`

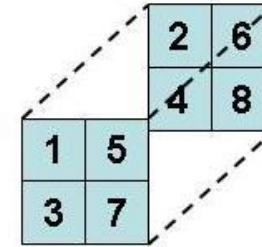
Fortran-style `f_style [[[1,5],
[3, 7]]
[[2, 6],
[4, 8]]]`

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

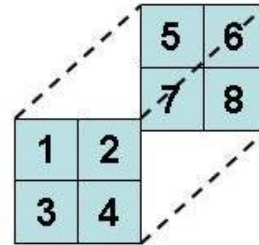
Reshape `f_style [[1, 5, 3, 7],
[2, 6, 4,8]]`



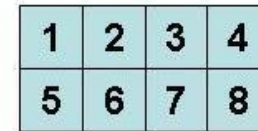
C - Style



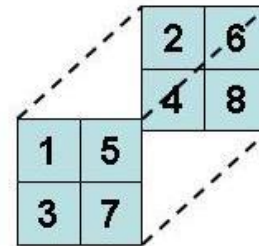
Fortran - Style



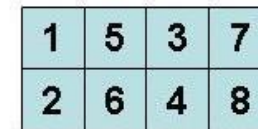
C - Style



Reshape



Fortran - Style



Reshape



Indexing - Slicing - Iteration

Array Indexing e Slicing

L'accesso agli elementi di un array avviene tramite l'operatore `[]`. Anche sugli array è applicabile l'operatore di slicing `[:]`.

Nel caso di array monodimensionali si adotta la stessa notazione delle liste.

Esempio

```
>>>a=ones(4)
```

```
>>>b=arange(1,5)
```

```
>>>a+=b
```

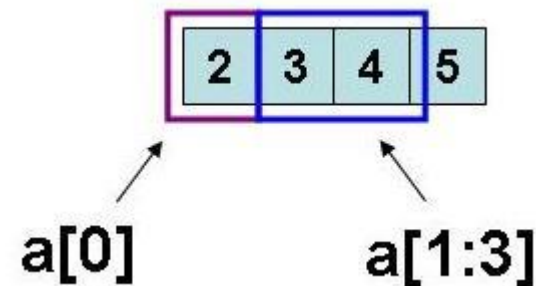
```
>>>print "a[0] ", a[0]
```

```
>>>2.0
```

```
>>>a[1:3]=a[1:3]*3
```

```
>>>print a
```

```
>>>[ 2.,  9., 12.,  5.]
```





Indexing – Slicing - Iteration

Nel caso di array multidimensionali:

Esempio

```
>>>a=array([[1,2,3],[4,5,6],[7,8,9],[10,11
```

```
>>> print a[0][0]
```

```
1
```

```
>>> print a[0,0]
```

```
1
```

```
>>> print a[2]
```

```
[7 8 9]
```

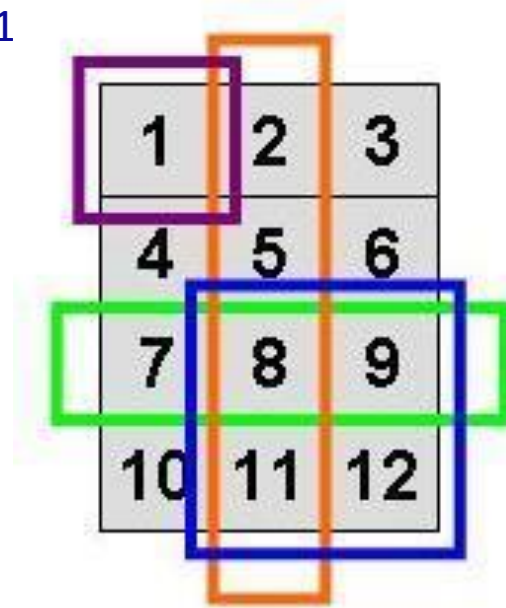
```
>>> print a[:,1]
```

```
a[ 2 , 5 , 8 , 11 ]
```

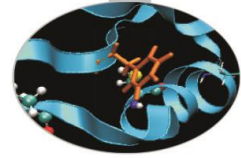
```
>>> print a[2:,1:3]
```

```
[[ 8 9]
```

```
[11 12]]
```



Indexing – Slicing – Iteration



```
>>> a=arange(25)
>>> a=a.reshape((5,5))
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]])
>>> a[:,1]
array([ 1,  6, 11, 16, 21])
>>> a[1]
array([5, 6, 7, 8, 9])
>>> a[1,:]
array([5, 6, 7, 8, 9])
>>> a[1,:]
array([5, 6, 7, 8, 9])
>>> a[1,::2]
array([5, 7, 9])
>>> a[1,5::-1]
array([9, 8, 7, 6, 5])
```



Array Copy

NOTA

Lo slicing per gli array è profondamente differente rispetto alla slicing per le liste. Nel caso di array il sotto-array generato mediante slicing è una reference all'area originale di memoria. Nel caso delle liste la sotto-lista è una copia per valore della lista originale.

```
>>>a=arange(6)
```

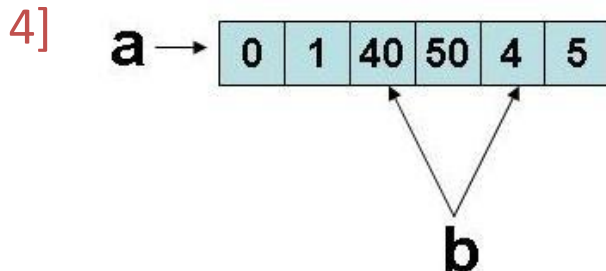
```
>>>b=a[2:5]
```

```
>>>b[0]=40
```

```
>>>b[1]=50
```

```
>>>print "a:", a , "b:", b
```

```
a: [ 0  1 40 50  4  5] b: [40 50  4
```



```
>>>a=range(6)
```

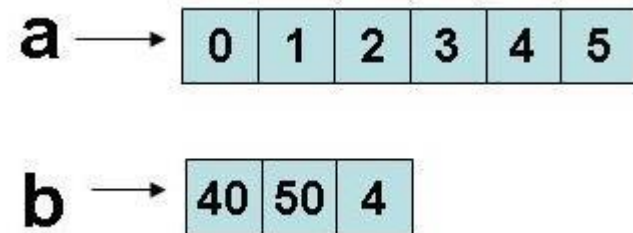
```
>>>b=a[2:5]
```

```
>>>b[0]=40
```

```
>>>b[1]=40
```

```
>>>print "a:", a , "b:", b
```

```
a: [ 0  1  2  3  4  5] b: [40 50
```





Array Copy

NOTA

Notiamo in questo caso che la copia avviene elemento per elemento e che i due oggetti sono distinti.

```
>>>a=arange(5)
```

```
>>>b=zeros_like(a)
```

```
>>>b[:]=a[:]
```

```
>>>b[3]=1000
```

```
>>>b==a
```

```
array([True,True,True,False,True],dtype=bool)
```



Array Copy

NOTA (copia per riferimento e per valore)

Nel caso degli array la copia è di default per referenza.

```
>>>a=arange(5)
```

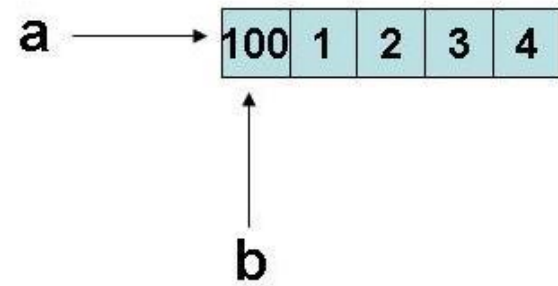
```
>>>b=a
```

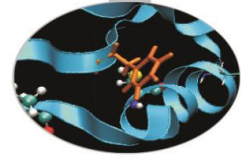
```
>>>b[0]=100
```

```
>>>print "a:", a , "b:" , b
```

```
a: [100,1,2,3,4]
```

```
b: [100,1,2,3,4]
```





Array Copy

Per effettuare un assegnamento per valore, si utilizza la funzione *copy*

```
>>>c=a.copy()
```

```
>>>print "id(c): ", id(c), "id(a):", id(a)
```

```
id(a): 18820584 id(c): 21335648
```

```
>>>c[0]=100
```

```
>>>print "c" , c , "a", a
```

```
c [100, 1, 2, 3, 4]  a [0, 1, 2, 3, 4]
```




Indexing – Slicing - Iteration

- La selezione degli elementi di un array può avvenire in modo più complesso:
 - Attraverso un array di indici
 - Tramite una maschera booleana

Questo metodo è noto come **fancy indexing**

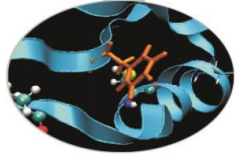
```
>>> x = np.arange(10,1,1)
array([10, 9, 8, 7, 6, 5, 4, 3, 2])
A=x[np.array([3,3,2,8])]
array([7, 7, 3, 2])
>>> AA=x[np.array([[3,3],[2,8]])]
array([[7, 7],
       [3, 2]])
```

Il nuovo array:

→ ha la shape dell'array di indici

→ ha tipo e valori dell'array di partenza

Indexing – Slicing - Iteration



Usando una maschera booleana

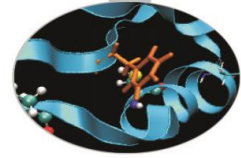
```
>>> y
```

```
array([[ 0,  1,  2,  3,  4,  5],  
       [ 6,  7,  8,  9, 10, 11],  
       [12, 13, 14, 15, 16, 17]])
```

```
>>> y[y>10]
```

```
array([11, 12, 13, 14, 15, 16, 17]) # 1d array
```

Indexing – Slicing - Iteration



Iteration

L'iterazione sugli elementi di un array può essere effettuata in diversi modi.

Attraverso il classico ciclo *for* lungo gli *assi* dell'array:

Esempio:

```
>>>a=arange(9)
```

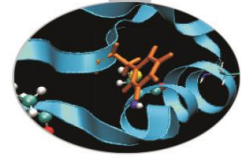
```
>>>a.shape=(3,3)
```

```
>>>for i in xrange(a.shape[0]):
```

```
    for j in xrange(a.shape[1]):
```

```
        a[i,j]=i+j
```

Il ciclo *for* applicato invece agli elementi di un array agisce di default sul primo asse.



Indexing – Slicing – Iteration

Esempio:

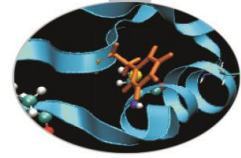
```
>>>for el in a:  
    print el  
  
>>>[ 0 1 2]  
    [ 1 2 3]  
    [ 2 3 4]
```

Attraverso l'iteratore *flat* su ogni elemento dell'array.

Esempio:

```
>>> for i in a.flat:  
    print i  
  
0 1 2 1 2 3 2 3 4
```

L'operazione di iterazione sugli array risulta tuttavia poco efficiente computazionalmente.
Per le operazioni sugli array Python dispone di funzioni scritte in C che operano direttamente sull'intero array...SEGUE



Operazioni numeriche su array

- Operazioni elementari con scalari:

```
>>> a = np.array([1, 2, 3, 4])
```

```
>>> a + 1           # somma di uno scalare  
array([2, 3, 4, 5])
```

```
>>> a - 2          # sottrazione di uno scalare  
array([-1, 0, 1, 2])
```

```
>>> 3*a            # moltiplicazione per uno scalare  
array([3, 6, 9, 12])
```

```
>>> 2**a           # potenza (base scalare)  
array([2, 4, 8, 16])
```

```
>>> a = np.array([1., 2., 3., 4.])
```

```
>>> a/2            # divisione per uno scalare  
array([0.5, 1., 1.5, 2.])
```



Operatori Aritmetici

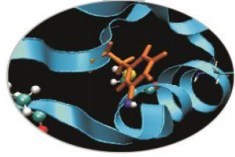
Operatori aritmetici

Gli operatori aritmetici agiscono in maniera *elementwise* sugli array.

Questa regola si applica sia ad operatori unari che ad operatori binari. E' inoltre valida per funzioni matematiche unarie (*sin*, *cos*, etc.)

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.ones(4) + 1
>>> a - b                # sottrazione tra array
array([-1., 0., 1., 2.])
>>> a*b                  # moltiplicazione tra array
array([2., 4., 6., 8.])
>>> a += 1; a           # autoincremento di array
array([2, 3, 4, 5])
>>> 2**(a+1) - a        # espressione con array
array([ 6, 13, 28, 59])
```

Operatori Aritmetici



Esempio

```
b=array([5,6,7,8])
```

```
c=arange(1,5)
```

```
d=c+b
```

```
print "Somma " ,b,"+",c, "= ", b+c
```

```
b+=1
```

```
print "Autoincremento b +=1  b=", b
```

```
print "Moltiplicazione c*3 " ,c, "* 3= ",c*3
```

```
print "Sin (c)", sin(c)
```

OUTPUT

Somma $[5,6,7,8] + [1,2,3,4] = [6,8,10,12]$

Autoincremento $b+=1$ $b = [6,7,8,9]$

Moltiplicazione $c*3$ $[1,2,3,4] *3 = [3,6,9,12]$

Sin(c) $[0.84147098, 0.90929743, 0.14112001, -0.7568025]$



Operatori di confronto

```
>>> a = np.array([1, 2, 3, 4])  
  
>>> b = np.array([4, 2, 2, 4])  
  
>>> a == b  
array([False, True, False, True])  
  
>>> a > b  
array([True, False, True, True])  
  
>>> c = np.array([1, 2, 3, 4])  
  
>>> np.array_equal(a,b)  
False  
  
>>> np.array_equal(a,c)  
True
```

Anche gli operatori standard di confronto lavorano *elementwise*

Per un confronto *arraywise* si usa il metodo `array_equal`

Operatori logici



```
>>> a = np.array([1, 1, 0, 0], dtype=bool)
```

```
>>> b = np.array([1, 0, 1, 0], dtype=bool)
```

```
>>> np.logical_or(a,b)  
array([True, True, True, False], dtype=bool)
```

```
>>> np.logical_and(a,b)  
array([True, False, False, False], dtype=bool)
```



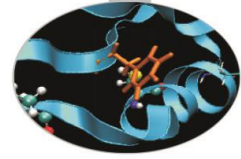
ufunc

- Numpy oltre alla definizione dell'oggetto `ndarray` definisce anche le funzioni universali `ufunc`
- Le funzioni `ufunc` permettono di operare elemento- elemento , sull'intero array senza dover usare dei loop espliciti.
- Queste funzioni sono dei wrapper a delle funzioni del core numpy tipicamente sviluppate in C o Fortran

```
>>>a=numpy.arange(100)
```

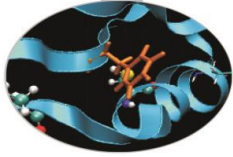
```
>>>b=cos(a)
```

ufunc



Esempio della funzione `ufunc_loop` definita nel core di Numpy

```
void ufunc_loop(void **args, int *dimensions, int *steps,
                void *data)
{
char *input_1 = (char*)args[0];
  char *input_2 = (char*)args[1];
  char *output = (char*)args[2];
  int i;
  for (i = 0; i < dimensions[0]; ++i) {
    *output = elementwise_function(*input_1, *input_2);
    input_1 += steps[0];
    input_2 += steps[1];
    output += steps[2];
  }
}
```



ufunc

- Ci sono più di 60 ufuncs
- Alcune `ufunc` sono nascoste dietro gli operatori aritmetici: i.e. `np.multiply(x,y)`
è chiamata quando si effettua l'operazione $a*b$
- NumPy offre funzioni trigonometriche, esponenziali and logaritmiche etc etc. Alcuni esempi:

```
>>> b = np.sin(a)
```

```
>>> b = np.arcsin(a)
```

```
>>> b = np.sinh(a)
```

```
>>> b = a**2.5 # power function
```

```
>>> b = np.log(a)
```

```
>>> b = np.exp(a)
```

```
>>> b = np.sqrt(a)
```

ufunc



- Funzione di confronto:

greater, less, equal, logical_and/_or/_xor/_nor, maximum, minimum, ...

```
>>> a = np.array([2, 0, 3, 5])
```

```
>>> b = np.array([1, 1, 1, 6])
```

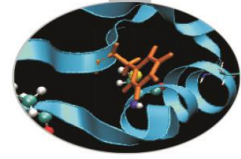
```
>>> np.maximum(a, b)
```

```
array([2, 1, 3, 6])
```

- Funzioni floating point:

floor, ceil, isreal, iscomplex, ...

Broadcasting



- Visto che tutte le tipologie di operatori tra array lavorano elementwise, gli array operandi devono avere (sempre) la stessa shape

```
>>> a = np.arange(4)
>>> a + np.array([1,2])
```

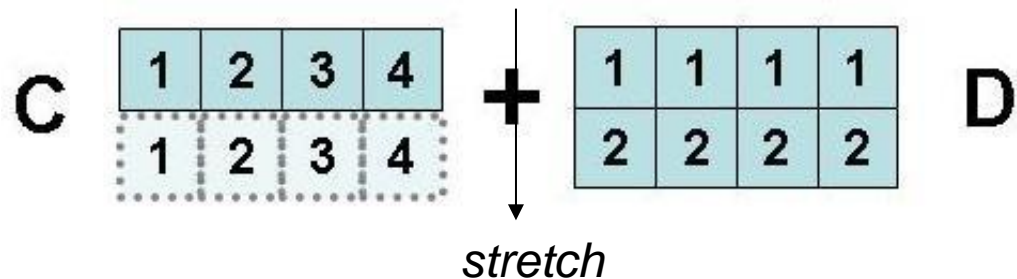
Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: operands could not be broadcast together with shapes (4) (2)

- E possibile talvolta operare con array che non hanno le stesse dimensioni

```
c=arange(1,5)
d=array([[1,1,1,1],[2,2,2,2]])
print d, "+", c "=", d+c
```

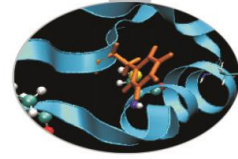


Questa modalità operativa viene definita di **broadcasting**



Broadcasting

- Il broadcasting quando applicabile permette di trattare con array che non hanno le stesse dimensioni.
- Il broadcasting segue due regole:
 - Se gli array non hanno lo stesso numero di dimensioni, l'array più piccolo viene ridimensionato (aggiungendo dimensione '1') fino a che entrambi gli array non hanno la stessa dimensione.
 - Array con dimensione '1' lungo una particolare direzione si comportano come l'array più grande lungo quella dimensione. Il valore è ripetuto lungo la direzione di broadcast.



Broadcasting

- Sugli array 1d si può sempre usare il broadcast.

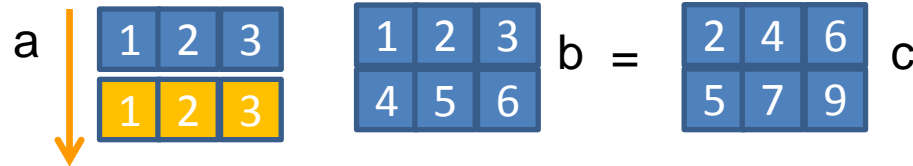
```
a=np.array([1,2,3])
```

```
a.shape # (3,)
```

```
b=np.array([[1,2,3],[4,5,6]])
```

```
b.shape #(2,3)
```

```
c=a+b #OK!! Broadcastable
```



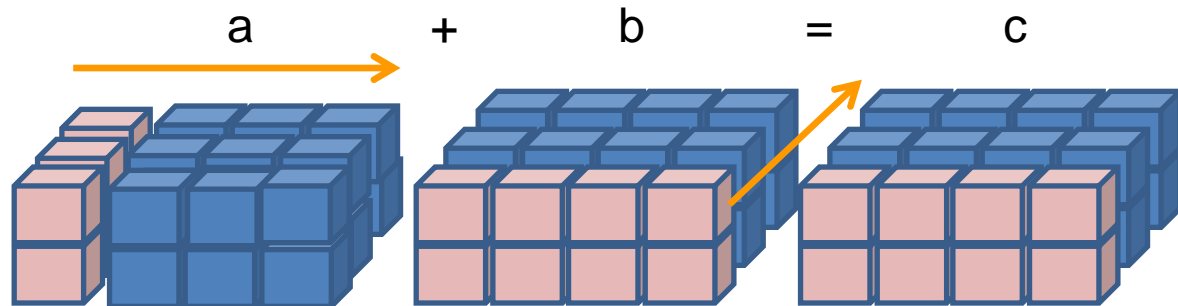
```
a=arange(6)
```

```
a=a.reshape((2,1,3))
```

```
b=arange(8)
```

```
b=b.reshape((2,4,1))
```

```
c=a+b #OK!! Broadcastable
```



```
a=a.arange(30)
```

```
a=a.reshape((2,5,3))
```

```
b=arange(8)
```

```
b=b.reshape((2,4,1)) #No Broadcastable
```




Array performance

- Le funzioni Numpy sono efficienti per lavorare sugli array, ma possono lavorare anche sugli scalari.
- Le funzioni contenute in *math* sono più efficienti sugli scalari rispetto alle funzioni Numpy.

```
>>> t=timeit.Timer('math.sin(math.pi)','import math')
```

```
>>> t.timeit()
```

```
0.20885155671521716
```

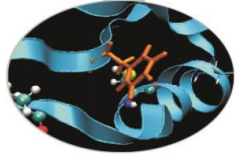
7 times faster

```
t=timeit.Timer('np.sin(np.pi)','import numpy as np')
```

```
>>> t.timeit()
```

```
1.3546336814836621
```

Efficienza di Calcolo



I cicli for non sono performanti in Python. Evitare di utilizzarli se non necessari!

```
def for_array(a):
```

```
    for i in xrange(a.shape[0]):
```

```
        for j in xrange(a.shape[1]):
```

```
            a[i,j]=3*a[i,j]+1
```

```
def no_loop(a):
```

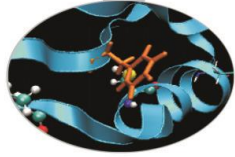
```
    a=a*3+1
```

Tempi di calcolo:

for_array on 1000 X 1000 array = 3.52 s

no_loop on 1000 X 1000 array = 0.015 s

Efficienza di Calcolo



Per la stessa motivazione le strutture dati *array* risultano più efficienti delle *liste*.

```
def somma_array(v1,v2):
```

```
    v=v1+v2
```

```
def somma_liste(l1,l2):
```

```
    l=[]
```

```
    for i in xrange(len(l1)):
```

```
        l.append(l1[i]+l2[i])
```

Tempo di calcolo:

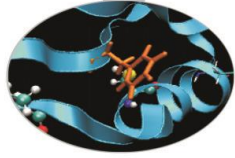
somma_array(v1,v2)

con 10^7 elementi = 0.05 s

somma_liste(l1,l2)

con 10^7 elementi = 10.49 s

Cenni di vettorizzazione



I cicli *for* sono piuttosto lenti in Python. Uno dei vantaggi nell'utilizzo degli array consiste nel fatto che molte operazioni possono essere svolte evitando loop espliciti questo procedimento prende il nome di *vettorizzazione*.

Esempi:

VECTORIZED VERSION

```
a=arange(0,4*pi,0.1)  
y=sin(a)*2
```

Anziché

SCALAR VERSION

```
y=zeros(len(a))  
for i in xrange(len(a)):  
    y[i]=sin(a[i])*2
```

In alcuni casi è necessario vettorizzare esplicitamente l'algoritmo:

- Direttamente: `vectorize(function)` # piuttosto lento!
- Manualmente: con tecniche opportune, p.e. slicing

Cenni di vettorizzazione



Solo in alcuni casi è possibile vettorizzare un'espressione:

ESEMPIO:

```
def func(x):
```

```
    if x<0: return 1
```

```
    else: return sin(x)
```

```
func(3)
```

```
func(array([1,-2,9]))
```

Traceback (most recent call last):

ValueError: The truth value of an array with more than one element is ambiguous. Use

`a.any()` or `a.all()`

Versione scalare per lavorare con gli array:

Cenni di vettorizzazione

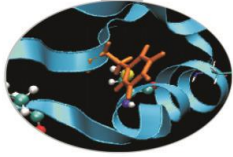


```
def func_NumPy(x):  
    r = x.copy() # allocate result array  
    for i in xrange(size(x)):  
        if x[i] < 0:  
            r[i] = 0.0  
        else:  
            r[i] = sin(x[i])  
    return r
```

- Implementazione penalizzante: molto lenta in Python
- Funziona solo per array monodimensionali

Utilizzo dello statement `where`

Cenni di vettorizzazione



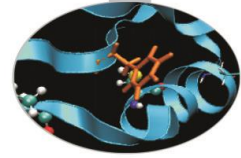
```
def f(x):  
    if condition:  
        x = <expression1>  
    else:  
        x = <expression2>  
    return x
```

```
def f_vectorized(x):  
    x1 = <expression1>  
    x2 = <expression2>  
    return where(condition, x1, x2)
```

Nel caso precedente

```
def func_NumPyV2(x):  
    return where(x < 0, 0.0, sin(x))
```

- Evito l'utilizzo di cicli for
- Funziona su strutture dati multidimensionali



Cenni di vettorizzazione

Lo slicing di array è spesso utilizzato per la vettorizzazione di operazioni. In ambito scientifico, per esempio, per applicazioni che riguardano schemi alle differenze finite o processamento di immagini è comune incontrare schemi del tipo:

$$x_{k=1} = x_{k-1} + 2x_k + x_{k+1} \quad k=1,2,\dots,n-1$$

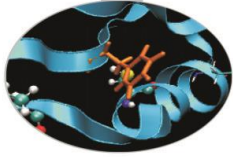
Che possono essere trattati tramite funzioni scalari con

```
for i in xrange(1,len(x)-1):  
    x[i]=x[i-1]+2*x[i]+x[i+1]
```

Oppure tramite vettorizzazione con:

$$x[1:n-1]=x[0:n-2]+2*x[1:n-1]+x[2:n]$$

Attributi e Metodi



L'object class *array* dispone di utili funzionalità implementate come propri metodi e attributi.

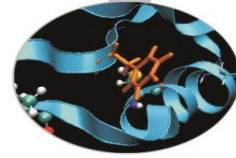
Attributi:

Gli attributi della classe *array* forniscono informazioni sulla struttura dell'array. Si ricordano:

- *dtype* tipo di dato
- *flat* array di (rank-1)
- *itemsize* e *nbytes* bytes usati da ogni singolo elemento e dall'intero array
- *ndim* numero di dimensioni dell'array
- *size* numero totale di elementi nell'array
- *shape* forma dell'array

Metodi

I metodi built-in implementano funzionalità che operano direttamente sull'array. Si ricordano:



Attributi e Metodi

take(indices, axis=None, out=None, mode='raise')

La funzione *take* estrapola un sottoarray costituito dagli elementi in posizione *indices* secondo l'asse *axis*

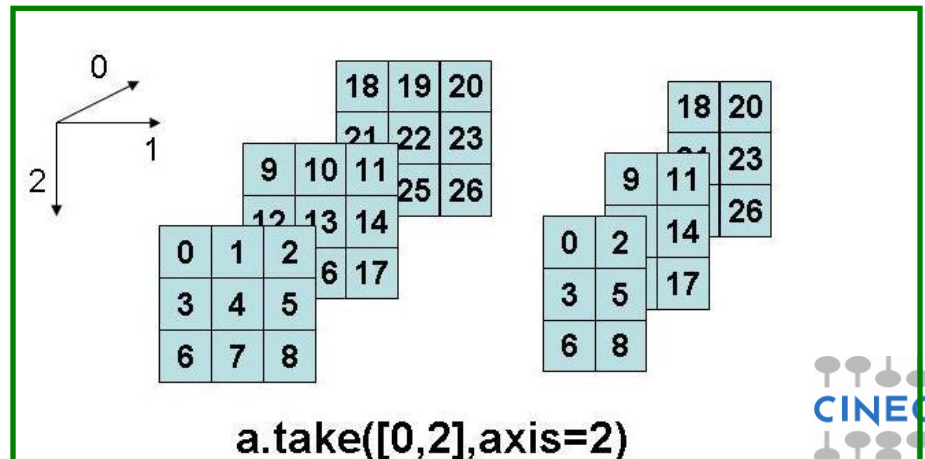
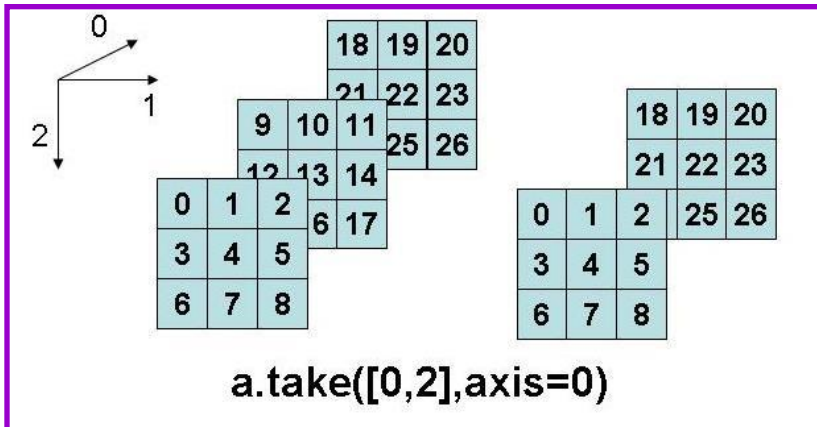
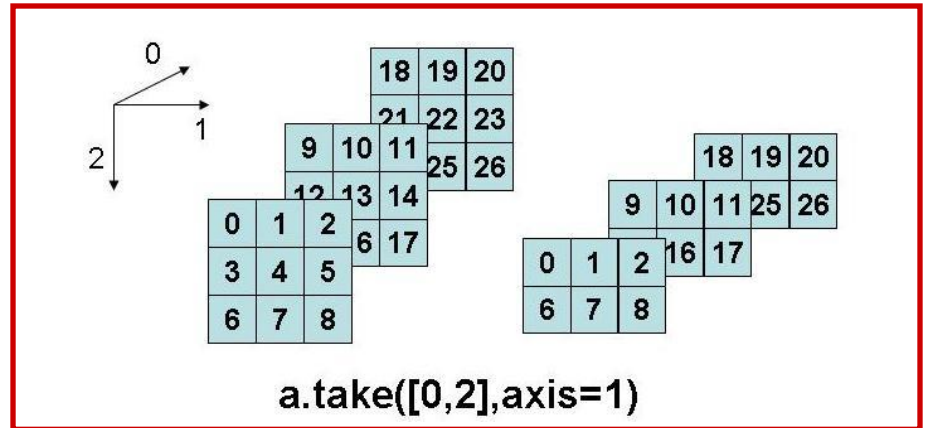
Esempio:

```
>>>a=arange(27)
```

```
>>>a.resize(3,3,3)
```

```
>>>y=a.take([0,2],axis=1)
```

```
>>>y2=a.take([0,2],axis=2)
```

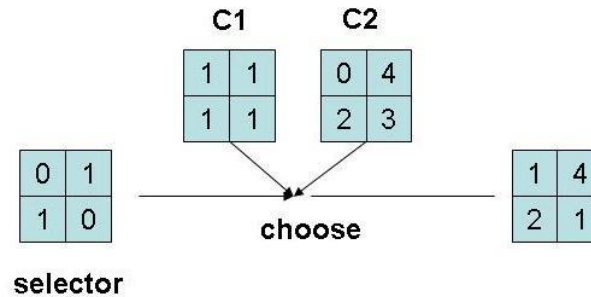




Attributi e Metodi

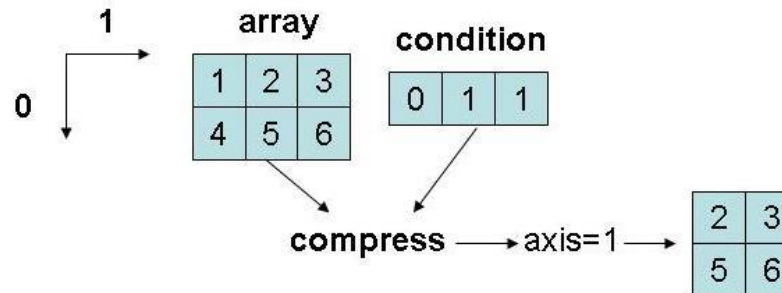
choose(choices, out=None, mode='raise')

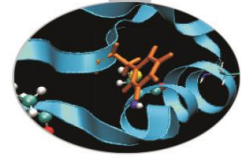
La funzione *choose* restituisce un array costruito dalle *choices* sulla base di un filtro selettore.



compress(condition, axis=None, out=None)

La funzione *compress* restituisce un array di elementi che soddisfano *condition* lungo l'asse *axis*





Attributi e Metodi

fill(value)

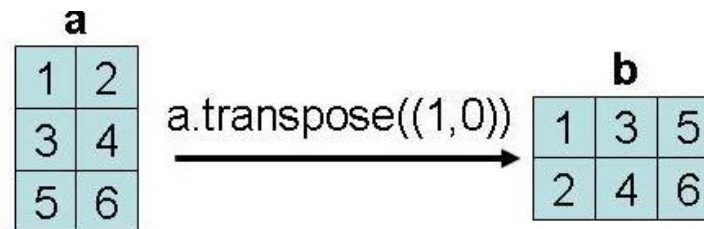
La funzione *fill* riempie l'array con il valore *value*.

sort(axis=-1, kind='quicksort', order=None)

La funzione *sort* riordina *inplace* i valori dell'array lungo *axis* con il metodo *kind*.

*transpose(*axis)*

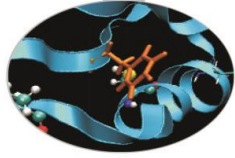
La funzione *transpose* traspone gli elementi dell'array secondo la permutazione specificata da *axis*.



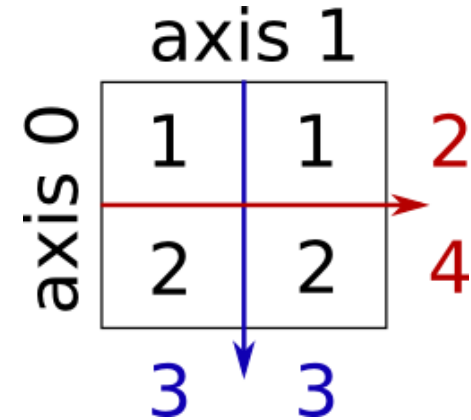
Per una lista completa di metodi e attributi si faccia riferimento a:

http://www.scipy.org/Numpy_Example_List

Operazioni di riduzione

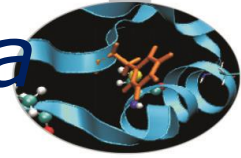


```
>>> x = np.array([[1, 1], [2, 2]])
>>> x.sum(axis=0) # somma per colonna
array([3, 3])
>>> x.sum(axis=1) # somma per riga
array([2, 4])
>>> x.sum()      # somma sull'intero array
6
```



- Tutte le altre operazioni di riduzione di array lavorano come la somma
- Sono disponibili operatori di riduzione di array di diverse tipologie:
- Statistica: `ndarray.mean()`, `ndarray.std()`, `ndarray.median()`, ...
- Calcolo di estremi: `ndarray.max()`, `ndarray.min()`, `ndarray.argmax()` – ritorna l'indice dell'elemento massimo -, `ndarray.argmin()`...
- Logiche: `ndarray.all()` – ritorna True se tutti gli elementi dell'array, o lungo un asse, sono True – e `ndarray.any()` – ritorna True se almeno un elemento dell'array, o di un suo asse, è True

Valutare una funzione 2D su griglia

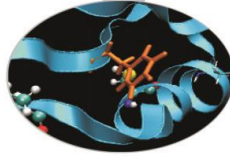


- Nelle scienze computazionali spesso capita di dover valutare una funzione su una griglia
- Data la griglia dei punti (x_i, y_i) con $x_i = [0, 1, 2, 3]$ e $y_i = [0, 1, 2, 3]$, vogliamo calcolare, per ciascun punto della griglia, il valore di una funzione $f(x, y)$
- Ingenuamente, potremmo pensare ad una soluzione NumPy come questa:

```
>>> x = np.arange(4)
>>> y = np.arange(4)
>>> def f(x,y):
...     return x**2+y
>>> f(x,y)
array([0, 2, 6, 12])
```

ma, il risultato è lungi da essere quello aspettato!

La soluzione: meshgrid

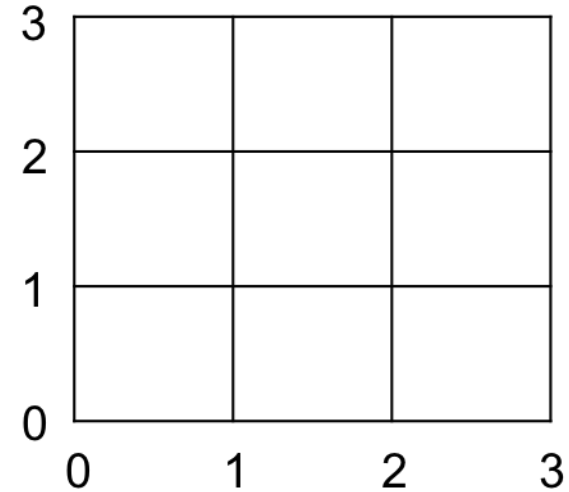


```
>>> x = np.arange(4)
>>> y = np.arange(4)
>>> xx, yy = np.meshgrid(x,y)

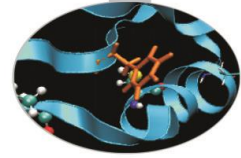
>>> xx
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])

>>> yy
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]])

>>> def f(x,y):
...     return x**2+y
```



```
>>> f(xx,yy)
array([[ 0,  1,  4,  9],
       [ 1,  2,  5, 10],
       [ 2,  3,  6, 11],
       [ 3,  4,  7, 12]])
```



I/O con Array Numpy

- Si possono usare le funzioni `eval` e `repr` per scrivere e leggere in formato ASCII

```
a = linspace(1, 21, 21)
```

```
a.shape = (2,10)
```

```
# ASCII format:
```

```
file = open('tmp.dat', 'w')
```

```
file.write('Here is an array a:\n')
```

```
file.write(repr(a)) # dump string representation of a
```

```
file.close()
```

```
# load the array from file into b:
```

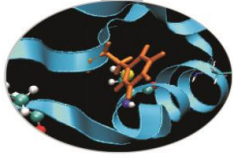
```
file = open('tmp.dat', 'r')
```

```
file.readline() # load the first line (a comment)
```

```
b = eval(file.read())
```

```
file.close()
```


I/O con Array Numpy



- L'I/O su file può essere anche gestito loadtxt e savetxt

Letture:

```
numpy.loadtxt(fname, dtype=<type  
'float'>, comments='#', delimiter=None, converters=None, skiprows=0,  
usecols=None, unpack=False, ndmin=0)
```

Scrittura:

```
numpy.savetxt(fname, X, fmt='% .18e', delimiter='  
' , newline='\n', header='', footer='', comments='#)
```



I/O con Array Numpy

- Text.txt

Student	Test1	Test2	Test3	Test4
Jane	98.3	94.2	95.3	91.3
Jon	47.2	49.1	54.2	34.7
Jim	84.2	85.3	94.1	76.4

```
>>>a = loadtxt('textfile.txt',skiprows=2,usecols=range(1,5))
```

```
>>>print a
```

```
[[ 98.3  94.2  95.3  91.3]
 [ 47.2  49.1  54.2  34.7]
 [ 84.2  85.3  94.1  76.4]]
```

```
>>>b = loadtxt('textfile.txt',skiprows=2,usecols=(1,-2))
```

```
>>> print b
```

```
[[ 98.3  95.3]
 [ 47.2  54.2]
 [ 84.2  94.1]]
```

I/O con Array Numpy



- Per lavorare con molti dati è più conveniente scrivere e leggere in formato binario.
- Il modo più semplice è usare il modulo *cPickle*

#Write to File

a1 and a2 are two arrays

```
import cPickle
```

```
file = open('tmp.dat', 'wb')
```

```
file.write('This is the array a1:\n')
```

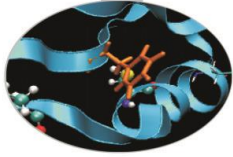
```
cPickle.dump(a1, file)
```

```
file.write('Here is another array a2:\n')
```

```
cPickle.dump(a2, file)
```

```
file.close()
```

I/O con Array Numpy



#Read from File

```
file = open('tmp.dat', 'rb')  
file.readline() # swallow the initial comment line  
b1 = cPickle.load(file)  
file.readline() # swallow next comment line  
b2 = cPickle.load(file)  
file.close()
```

Il modulo cPickle garantisce I/O più rapido e un metodo di immagazzinamento dati a minor costo.

I/O con Array Numpy



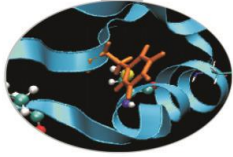
- NumPy prevede anche un proprio formato, non portabile, ma piuttosto efficiente nelle operazioni di I/O

Scrittura e lettura binaria nel formato NumPy

```
>>> my_data = np.ones((3,3))  
  
>>> np.save('myData.npy', my_data)  
  
>>> r_data = np.load('myData.npy')
```

In Python è possibile eseguire operazioni di I/O nei formati binari più diffusi in ambito tecnico scientifico:

- Il supporto al formato HDF5 è disponibile nel modulo ad-hoc *h5py* (<http://code.google.com/p/h5py/>) ed in *PyTables*
- *SciPy* dispone del supporto per l'I/O nei formati *NetCDF*, *Matlab* e *MatrixMarket*



Matrix

- Numpy fornisce delle classi standard che ereditano da array e che usano la sua struttura interna.
- *Matrix* eredita da *ndarray* → stessi metodi e attributi
- La classe *Matrix* ha degli attributi speciali
 - .T trasposta
 - .H coniugata trasposta
 - .I inversa
 - .A array bidimensionale
- *Matrix* definisce oggetti esclusivamente bidimensionali
- *Matrix* ridefinisce l'operatore * per la moltiplicazione matriciale.
- Gli oggetti *Matrix* hanno la precedenza rispetto agli array semplici



Matrix

```
>>>import numpy as np
```

```
>>>a=np.arange(16)
```

```
>>>a=a.reshape((4,4))
```

```
>>>b=2*np.arange(16)
```

```
>>> b=b.reshape((4,4))
```

```
>>>c=a*b      #element by element
```

```
>>> ma=np.matrix(a)
```

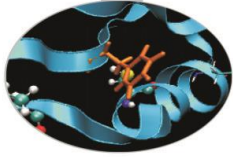
```
>>> mb=np.matrix(b)
```

```
>>> mc=ma*mb   #matrixmul
```

```
>>>mmc=ma*b   #matrixmul
```

```
array([[ 0,  2,  8, 18],  
       [ 32, 50, 72, 98],  
       [128, 162, 200, 242],  
       [288, 338, 392, 450]])
```

```
matrix([[ 112, 124, 136, 148],  
        [ 304, 348, 392, 436],  
        [ 496, 572, 648, 724],  
        [ 688, 796, 904, 1012]])
```



linalg

- Il modulo Numpy contiene oltre alla definizione dell'object array anche alcuni moduli.

`linalg`

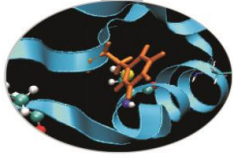
Il modulo `linalg` è un modulo che contiene alcuni algoritmi di algebra lineare all'interno di numpy.

Contiene funzioni per risolvere sistemi lineari, calcolare autovalori e autovettori, fattorizzazioni, inverse di matrici, prodotto matriciale.

```
>>> dir(linalg)
```

```
['LinAlgError', 'Tester', '__builtins__', '__doc__', '__file__', '__name__',  
'__package__', '__path__', 'bench', 'cholesky', 'cond', 'det', 'eig', 'eigh',  
'eigvals', 'eigvalsh', 'info', 'inv', 'lapack_lite', 'linalg', 'lstsq', 'matrix_power',  
'matrix_rank', 'norm', 'pinv', 'qr', 'slogdet', 'solve', 'svd', 'tensorinv',  
'tensorsolve', 'test']
```

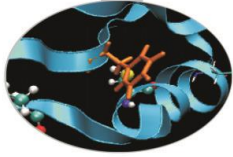

linalg



Esempio:

```
A = np.zeros((10,10)) # arrays initialization
x = np.arange(10)/2.0
for i in range(10):
...   for j in range(10):
...     A[i,j] = 2.0 + float(i+1)/float(j+i+1)
b = np.dot(A, x)
y = np.linalg.solve(A, b) # A*y=b → y=x
# eigenvalues only:
>>> A_eigenvalues = np.linalg.eigvals(A)
# eigenvalues and eigenvectors:
>>> A_eigenvalues, A_eigenvectors = np.linalg.eig(A)
```

random



`random` è un altro modulo definito dentro `numpy` per la generazione di numeri casuali

`dir(random)`

```
['RandomState', 'Tester', '__RandomState_ctor', '__all__', '__builtins__',  
'__doc__', '__file__', '__name__', '__package__', '__path__', 'bench', 'beta',  
'binomial', 'bytes', 'chisquare', 'dirichlet', 'exponential', 'f', 'gamma',  
'geometric', 'get_state', 'gumbel', 'hypergeometric', 'info', 'laplace', 'logistic',  
'lognormal', 'logseries', 'mtrand', 'multinomial', 'multivariate_normal',  
'negative_binomial', 'noncentral_chisquare', 'noncentral_f', 'normal', 'np',  
'pareto', 'permutation', 'poisson', 'power', 'rand', 'randint', 'randn', 'random',  
'random_integers', 'random_sample', 'ranf', 'rayleigh', 'sample', 'seed',  
'set_state', 'shuffle', 'standard_cauchy', 'standard_exponential',  
'standard_gamma', 'standard_normal', 'standard_t', 'test', 'triangular',  
'uniform', 'vonmises', 'wald', 'weibull', 'zipf']
```

random



Generare un array di numeri casuali usando il modulo standard numpy è inefficiente, meglio usare il modulo numpy.random

```
>>> np.random.seed(100)
```

```
>>> x = np.random.random(4)
```

```
array([ 0.89132195,  0.20920212,  0.18532822,  
        0.10837689])
```

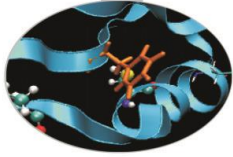
```
>>> y = np.random.uniform(1, 1, n) # n uniform  
numbers in interval (1,1)
```

Distribuzione normale

```
>>> mean = 0.0; stdev = 1.0
```

```
>>> u = np.random.normal(mean, stdev, n)
```

Scipy



- Special Functions ([scipy.special](#))
- Signal Processing ([scipy.signal](#))
- Fourier Transforms ([scipy.fftpack](#))
- Optimization ([scipy.optimize](#))
- General plotting ([scipy.\[plt, xplt, gplt\]](#))
- Numerical Integration ([scipy.integrate](#))
- Linear Algebra ([scipy.linalg](#))
- Input/Output ([scipy.io](#))
- Genetic Algorithms ([scipy.ga](#))
- Statistics ([scipy.stats](#))
- Distributed Computing ([scipy.cow](#))
- Fast Execution ([weave](#))
- Clustering Algorithms ([scipy.cluster](#))
- Sparse Matrices* ([scipy.sparse](#))