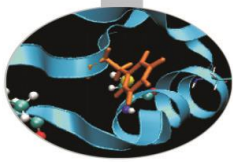
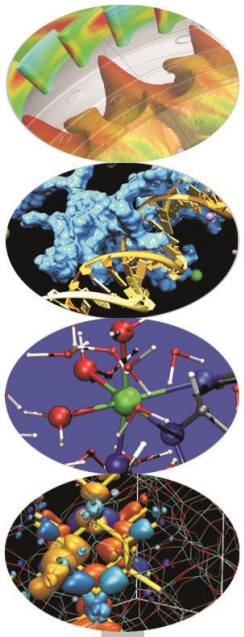
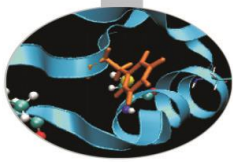


Python per il calcolo scientifico



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.

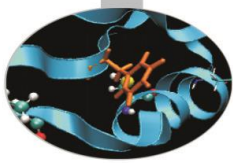
Oltre alla standard library sono stati sviluppati moduli ad hoc per il calcolo scientifico (Numpy, Pylab, Scipy…).

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 Elettormagnetiche)
- 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:

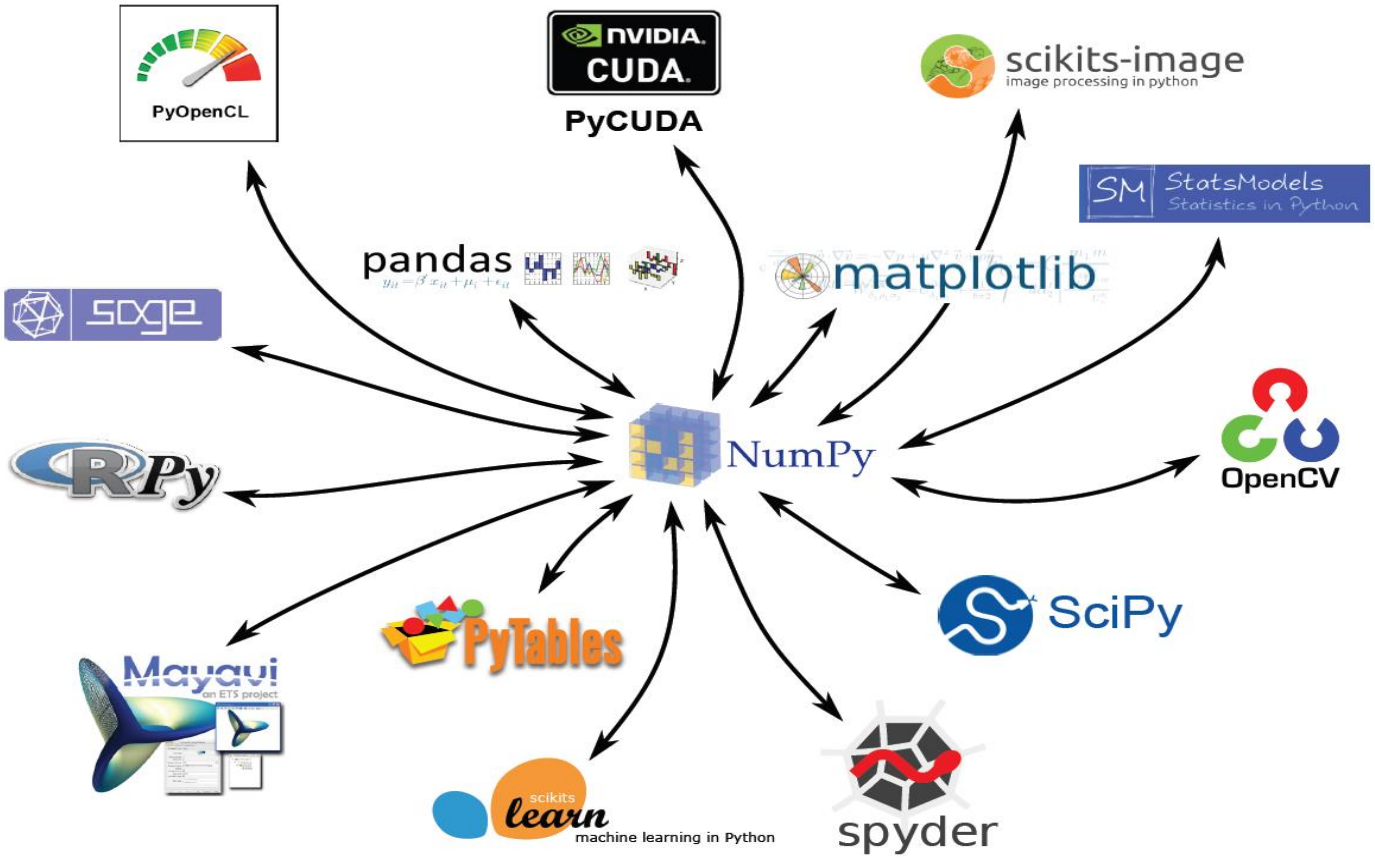
  http://numpy.scipy.org/

- Introduzione al modulo *pylab:* creazione di grafici

  Official web-site:

  http://matplotlib.sourceforge.net/

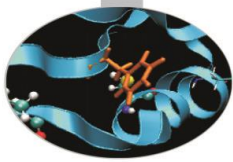
Numpy



“Life is too short to write C++ code“

David Beazley - EuroScipy 2012 Bruxelles

Modulo 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.

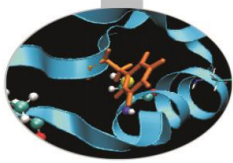
Teminologia:

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.

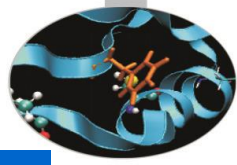
Import del modulo



- Importiamo il modulo come di consueto

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

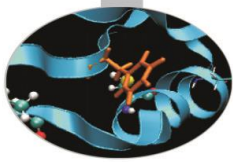
Numpy fornisce un nuovo tipo di dato: un array N-dimensionale (ndarray).



Organizzazione del modulo

Sub-Packages	Purpose	Comments
core	basic objects	all names exported to numpy
lib	Addintional 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

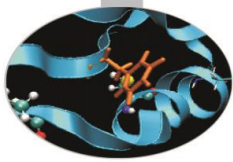
ndarray



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;
```

Creazione di un Array



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

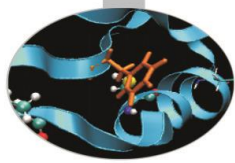
1	4	2	5	3	6

Fortran - Style

1	2	3	4	5	6

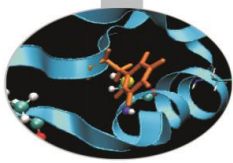
C-Style

Creazione di un Array



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

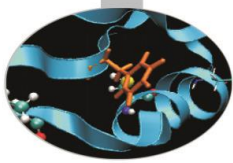
```
>>>import numpy as np
>>>a=np.array([1,2,3,4])
>>>lista1=[1,2,3,4]
>>>tupla=(5,6,7,8)
>>>a=np.array(lista)       #from a list
>>>b=np.array(tupla)      #from a tupla
>>>c=np.array([lista,tupla]) #from a list and from a tupla
>>>a.dtype
dtype('int32')
```



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

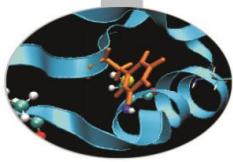


ndtype

- 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 */
npy_intp *dimensions;      /* size in each dimension */
```

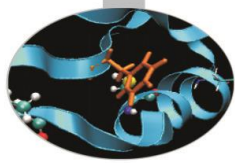
- L'attributo *shape* specifica la forma dell'array:

```
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
```

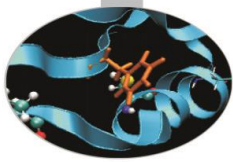
itemsize



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)
```

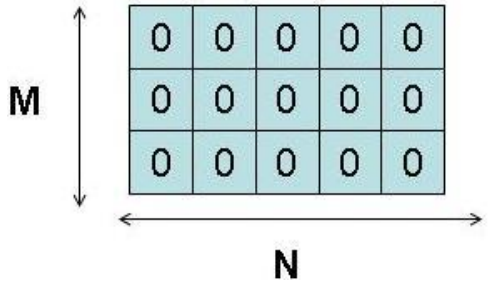
Creazione di un Array



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

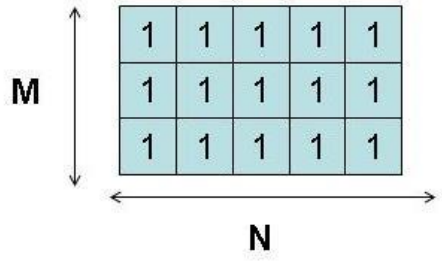
- La funzione *zeros* crea un array di dimensioni *shape* di soli zeri.

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

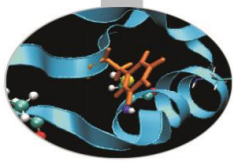


- La funzione *ones* crea un array di dimensione *shape* di soli uni.

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

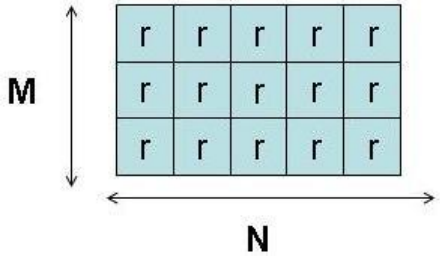


Creazione di un Array



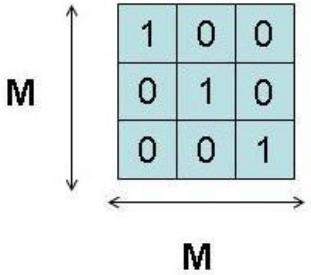
- La funzione *empty* crea un array di dimensioni *shape* senza inizializzazione.

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



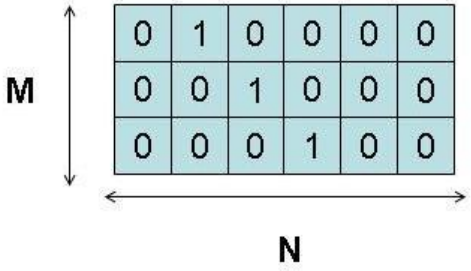
- La funzione *identity* genera la matrice identità *n X n*

  identity(n, dtype=None)

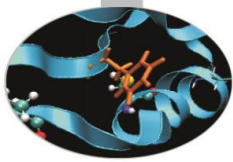


- La funzione *eye* crea una matrice N X M riempendo di 1 la k-esima diagonale

  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.

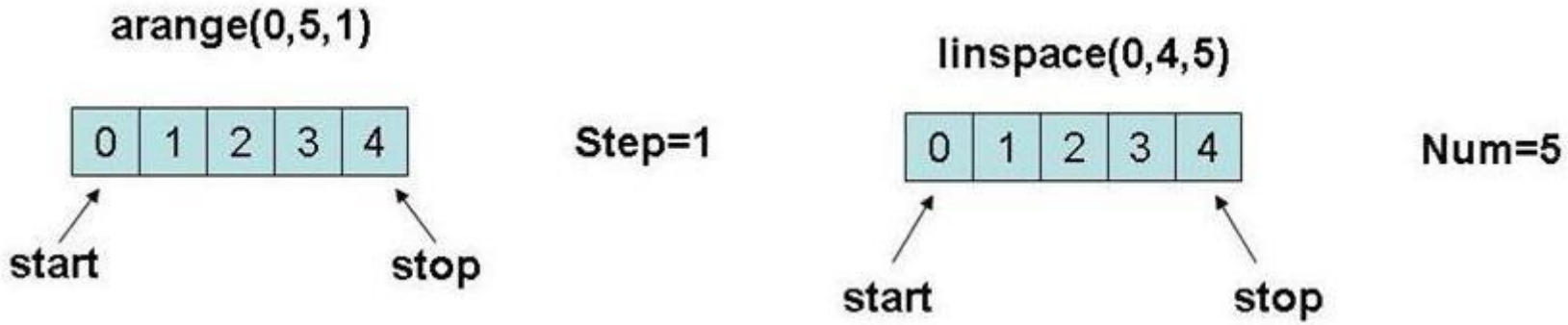
arange([start,] stop[, step,], dtype=None)

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

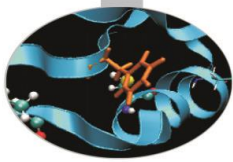
In alternativa:

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

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



Creazione di un Array

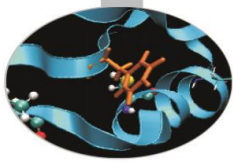


- E’ possibile creare array anche da stringhe con la funzione fromstring

```
>>> 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…

Reshaping & Resizing



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

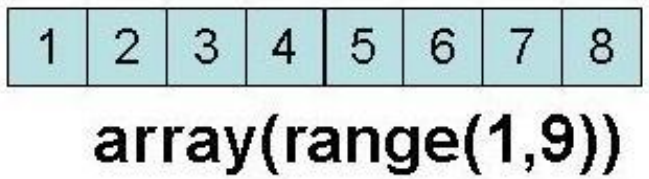
```
>>>a=arange(20)
>>>a.resize(5,6)          #Ok
```

Reshape & Resize

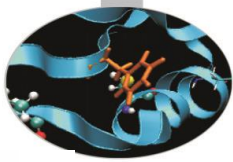
```
>>>b=a
>>>a.resize(3,3)                    #Error a is referenced by b
Traceback (most recent call last):
  File "<pyshell#160>", line 1, in <module>
    a.resize(3,3)
ValueError: cannot resize an array that has been referenced or is referencing
another array in this way.  Use the resize function
```

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
```



Reshape & Resize



OUTPUT

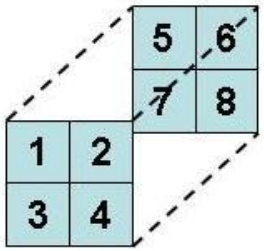
Shape (8,)

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

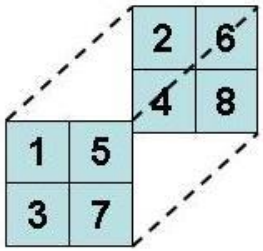
Fortran-style f_style [[[1,5],
[3, 7]]
[[2, 6],
[4, 8]]]

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

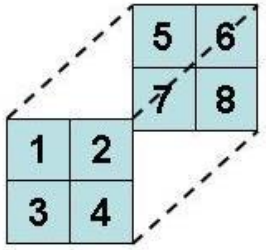
Reshape f_style [[1, 5, 3, 7],
[2, 6, 4,8]]



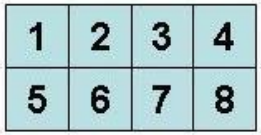
C - Style



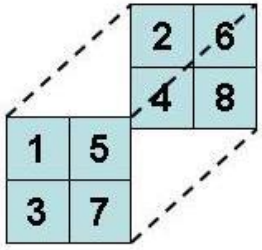
Fortran - Style



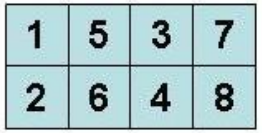
C - Style



Reshape

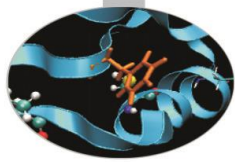


Fortran - Style



Reshape

Indexing - Slicing - Iteration



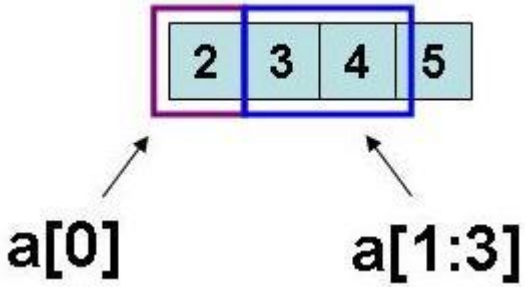
Array Indexing e Slicing

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

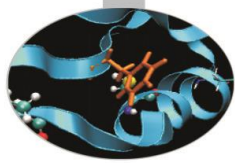
Nel caso di array monodimensionali si adotta la stessa notazione delle liste.

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.]
```



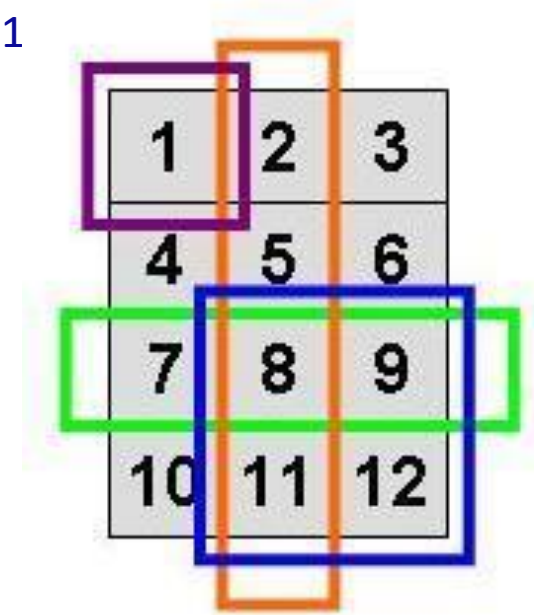
Indexing – Slicing - Iteration



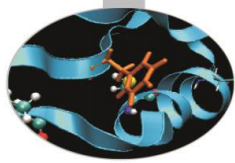
Nel caso di array multidimensionali:

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]]
```

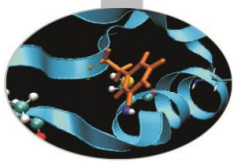


Indexing – Slicing - Iteration



```
>>> 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 Selection

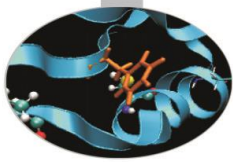


- La selezione degli elementi di un array può avvenire in modo più complesso:
  - Attraverso un array di indici
  - Tramite una maschera booleana

```
>>> 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

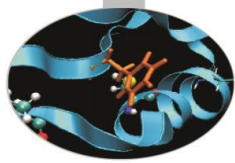
Array Selection



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.

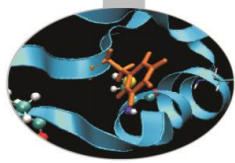
Attraverso il classico ciclo *for* lungo gli *assi* dell'array:

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]
```

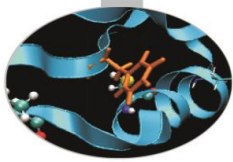
Attraverso l'iteratore *flat* su ogni elemento dell'array.

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

Operatori Aritmetici



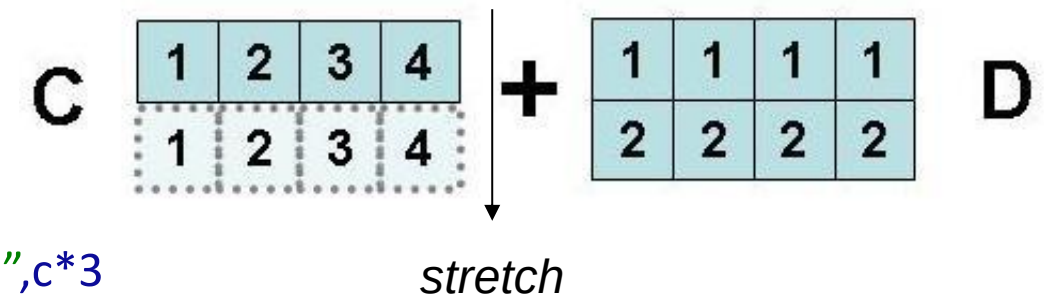
Operatori aritmetici

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

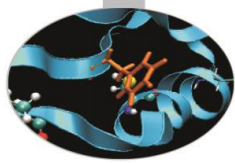
Questa regola si applica sia ad operatori unari che ad operatori binari. E' inoltre valida per funzioni matematiche unarie (*sin*, *cos*, etc.)

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)
```



Operatori Aritmetici



OUTPUT

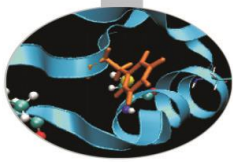
Somma [5,6,7,8] + [1,2,3,4] = [6,8,10,12]

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]

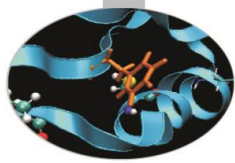
ufunc



- 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)
```

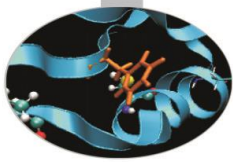
ufunc



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



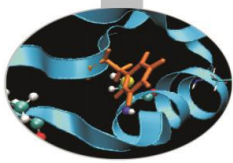
- Ci sono più di 60 ufuncs
- Alcune `ufunc` sono nascoste dietro gli operatori aritmetici: i.e. np.multiply(x,y)

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:

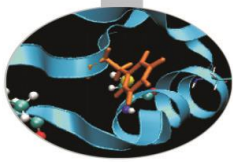
greater, less, equal, logical_and/_or/_xor/_nor, maximum, minimum, ...

```
>>> 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:

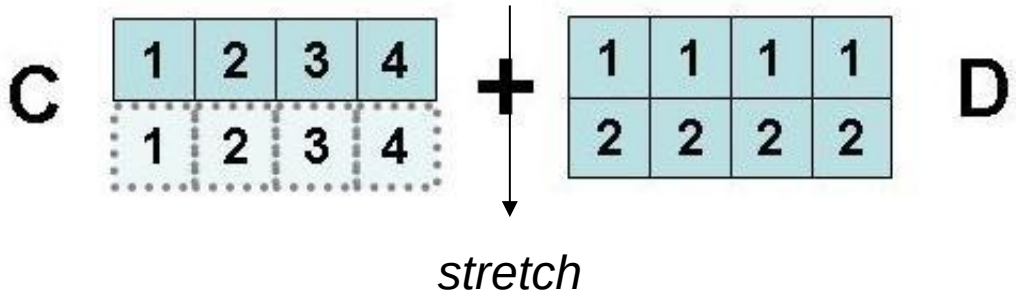
floor, ceil, isreal, iscomplex, ...

Broadcasting



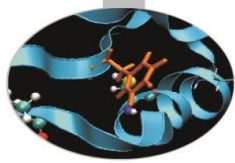
- E possibile talvolta operare con array che non hanno le stesse dimensioni

```
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.

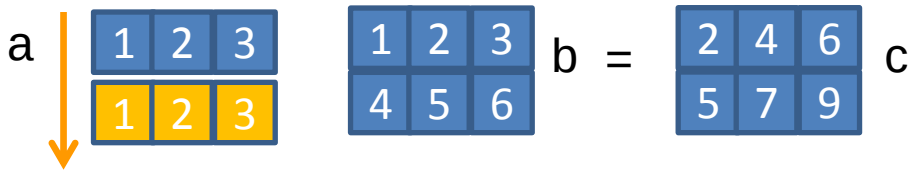
- Il broadcasting segue due regole:

  - 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.

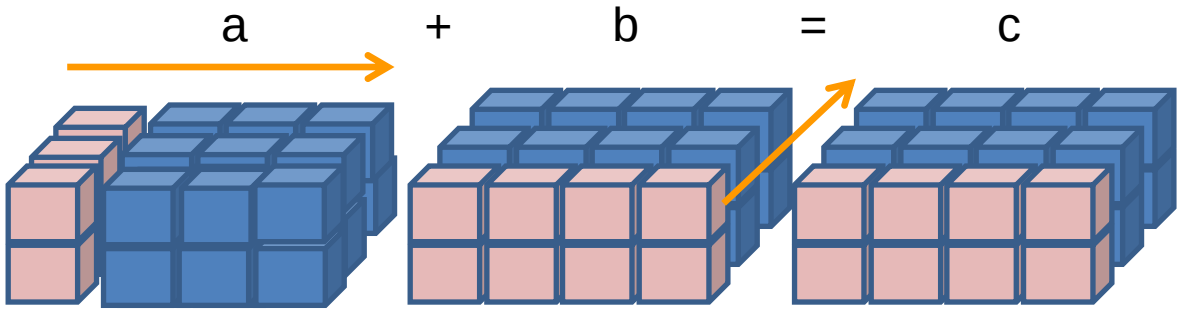
Broadcasting

- 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
```

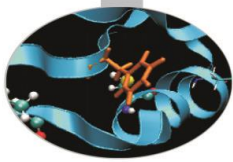



```
a=arange(6)
a=a.reshape((2,1,3))
b=arange(8)
b=b.reshape((2,4,1))
c=a+b  #OK!! Broadcastable
```




```
a=a.arange(30)
a=a.reshape((2,5,3))
b=arange(8)
b=b.reshape((2,4,1)) #No Broadcastable
```

Array performance

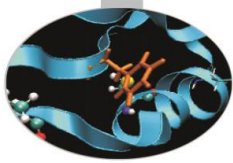


- Le funzioni Numpy sono efficienti per lavorare sugli array, ma possono lavorare anche sugli scalari.
- Le funzioni contenute in *math* sono più efficienti **sugli scalari** rispetto alle funzioni Numpy.

```
>>> t=timeit.Timer('math.sin(math.pi)','import math')
>>> t.timeit()
0.20885155671521716
t=timeit.Timer('np.sin(np.pi)','import numpy as np')
>>> t.timeit()
1.3546336814836621
```

7 times faster

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

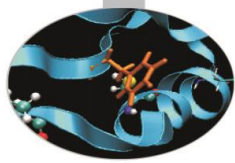
SCALAR VERSION

```
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:

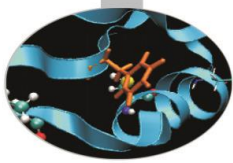
ESEMPIO:

```
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

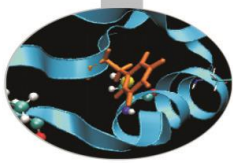


```
def func_NumPy(x):
        r = x.copy() # allocate result array
        for i in xrange(size(x)):
                if x[i] < 0:
                        r[i] = 0.0
                else:
                        r[i] = sin(x[i])
                return r
```

- 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
```

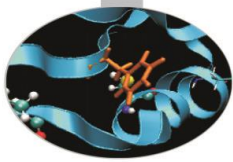
```
def f_vectorized(x):

        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:

$$x_{k=}x_{k-1} + 2x_k + x_{k+1} \qquad k=1,2,\ldots,n-1$$

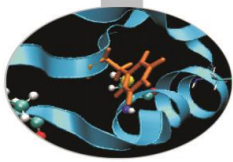
Che possono essere trattati tramite funzioni scalari con

```
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]
```

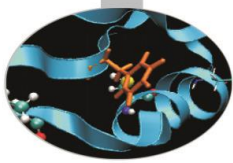
Efficienza di Calcolo



I cicli for non sono perfomanti 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

def for_mono(a):
    for i in xrange(a.size):
        a[i]=a[i]*3+1
```

Efficienza di Calcolo



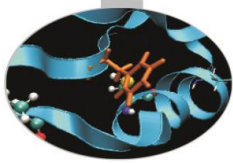
Tempi di calcolo:

for_array	on 1000 X 1000 array	= 3.52 s
no_loop	on 1000 X 1000 array	= 0.015 s
for_mono	on 10000000 array	= 33.23 s
no_loop	on 10000000 array	= 0.27 s

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])
```

Efficienza di calcolo

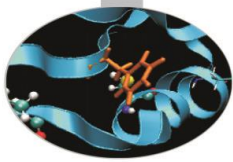


Tempo di calcolo:

somma_array(v1,v2) con 10^7 elementi = 0.05 s

somma_liste(l1,l2) con 10^7 elementi = 10.49 s

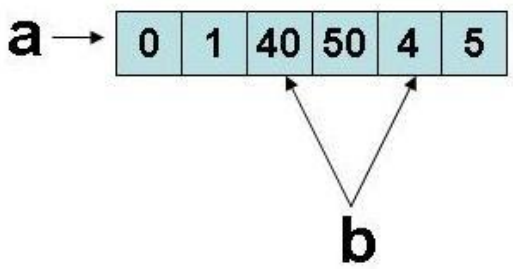
Indexing – Slicing - Iteration



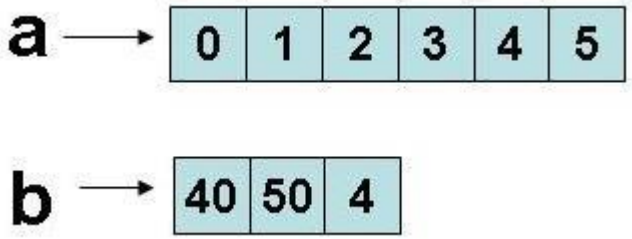
NOTA

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.

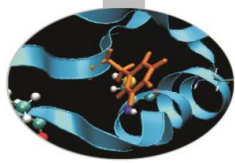
```
>>>a=arange(6)
>>>b=a[2:5]
>>>b[0]=40
>>>b[1]=50
>>>print "a:", a , "b:", b
a: [ 0  1 40 50  4  5]  b: [40 50  4]
```



```
>>>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
```



Indexing – Slicing - Iteration

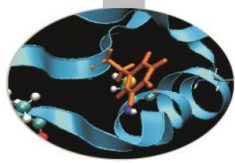


NOTA

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

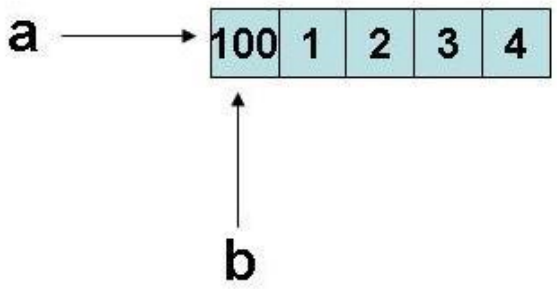


NOTA (copia per riferimento e per valore)

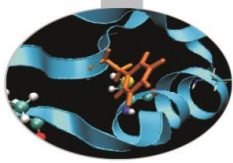
Nel caso degli array la copia è di default per referenza.

```
>>>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]



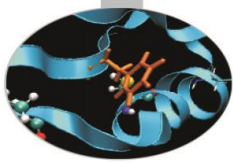
Array Copy



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]
```

Attributi e Metodi



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

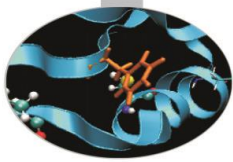
Attributi:

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:



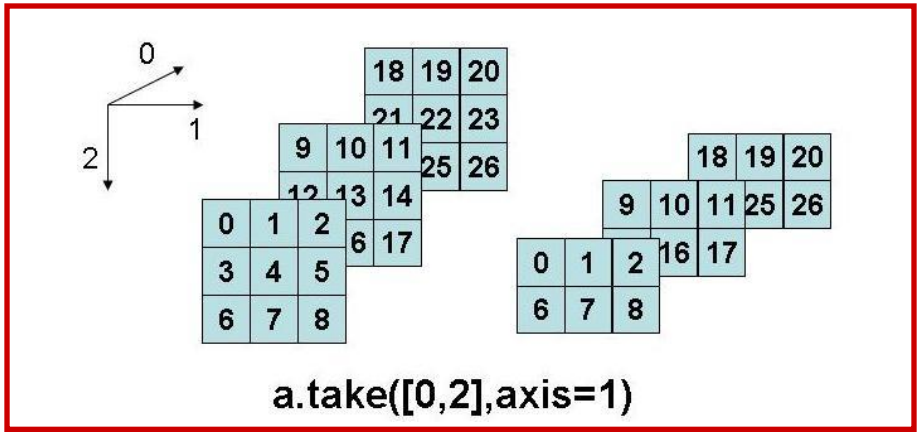
Attributi e Metodi

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

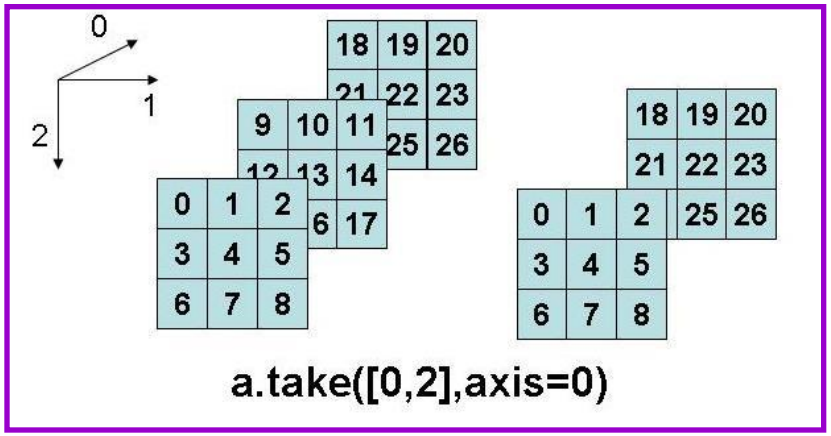
La funzione *take* estrapola un sottoarray costituito dagli elementi in posizione *indices* secondo l'asse *axis*

Esempio:

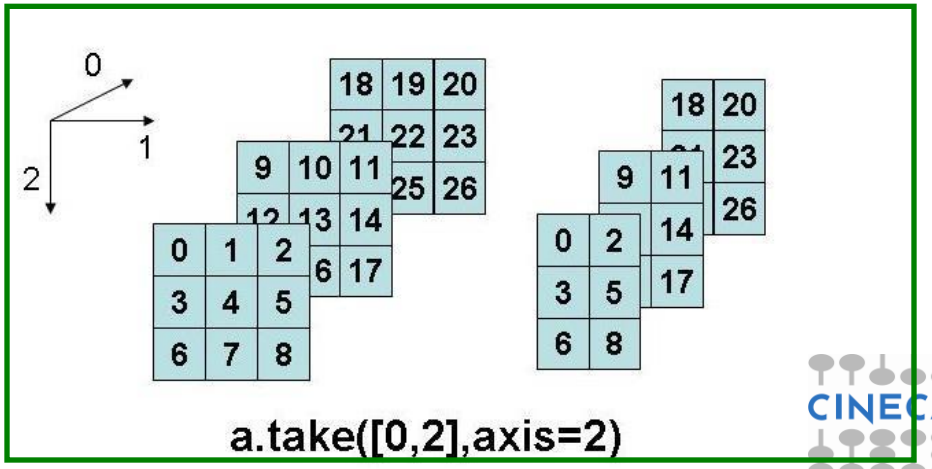
```
>>>a=arange(27)
>>>a.resize(3,3,3)
>>>y=a.take([0,2],axis=1)
>>>y2=a.take([0,2],axis=2)
```



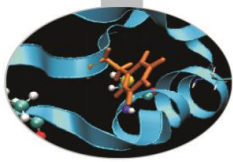
a.take([0,2],axis=1)



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



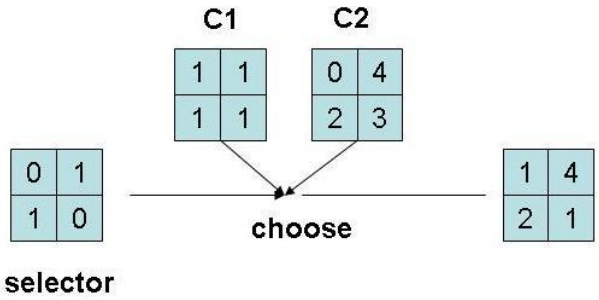
a.take([0,2],axis=2)



Attributi e Metodi

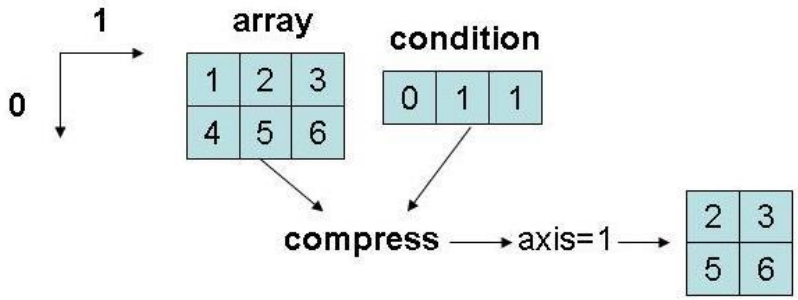
choose(choices, out=None, mode='raise')

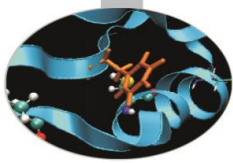
La funzione *choose* restituisce un array costruito dalle *choices* sulla base di un filtro selettore.



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)

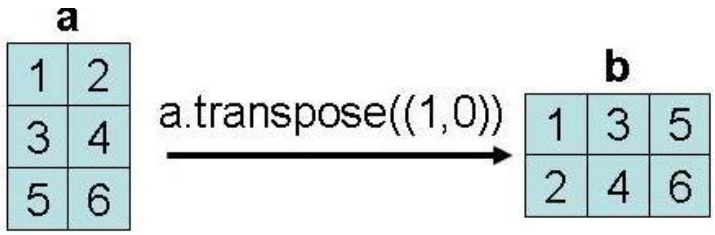
La funzione *fill* riempe l'array con il valore *value.*

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

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

*transpose(*axis)*

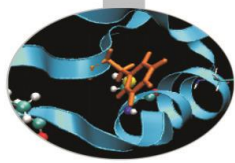
La funzione *transpose* traspone gli elementi dell'array secondo la permutazione specificata da *axis.*



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

http://www.scipy.org/Numpy_Example_List

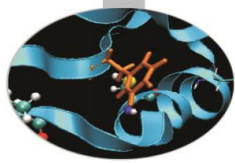
I/O con Array Numpy



- 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

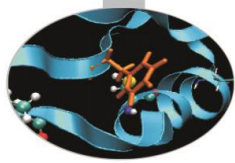
Lettura:

```
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
```

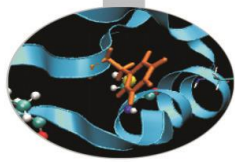
```
[[ 98.3  94.2  95.3  91.3]
 [ 47.2  49.1  54.2  34.7]
 [ 84.2  85.3  94.1  76.4]]
```

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

```
>>> print b
```

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

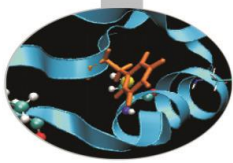
I/O con Array Numpy



- 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()
```

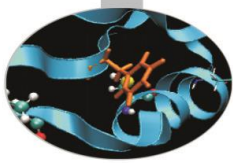
I/O con Array Numpy



```
#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.

Matrix

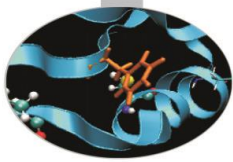


- Numpy fornisce delle classi standard che ereditano da array e che usano la sua struttura interna.

- *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

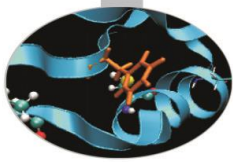


```
>>>import numpy as np
>>>a=np.arange(16)
>>>a=a.reshape((4,4))
>>>b=2*np.arange(16)
>>> b=b.reshape((4,4))
>>>c=a*b            #element by element
>>> ma=np.matrix(a)
>>> mb=np.matrix(b)
>>> mc=ma*mb       #matrixmul
>>>mmc=ma*b       #matrixmul
```

```
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`

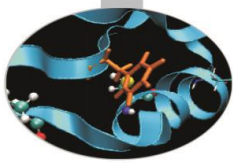
Il modulo `linalg` è un modulo che contiene alcuni algoritmi di algebra lineare all'interno di numpy.

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']

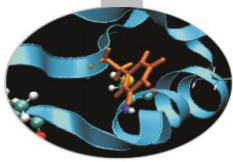
linalg



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

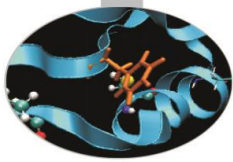


`random` è un altro modulo definito dentro numpy per la generazione di numeri casuali

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



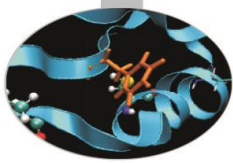
Generare un array di numeri casuali usando il modulo standard numpy è inefficiente, meglio usare il modulo numpy.random

```
>>> 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

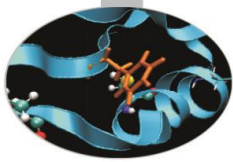
```
>>> mean = 0.0; stdev = 1.0
>>> u = np.random.normal(mean, stdev, n)
```

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)

Matplotlib: Modulo Pylab



Uno strumento per la grafica bidimensionale è fornito dalla libreria Matplotlib.

La libreria Matplotlib è una libreria che nasce in origine per emulare in ambiente Python i comandi grafici di Matlab.

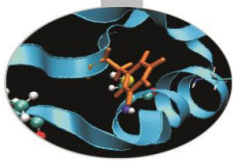
Matplotlib è completamente sviluppata in Python e utilizza il modulo Numpy per la rappresentazione di grandi array.

Matplotlib è divisa in tre parti:

- Pylab interface: set di funzioni fornite dal modulo Pylab.
- Matplotlib API
- Backend: grafici per l'output su file e visuali per l'output su interfaccie grafiche.

La libreria Matplotlib è particolarmente indicata per il calcolo scientifico.

Contiene diverse funzioni in tal senso. Inoltre è possibile utilizzare la sintassi LaTex per aggiungere formule sui grafici.



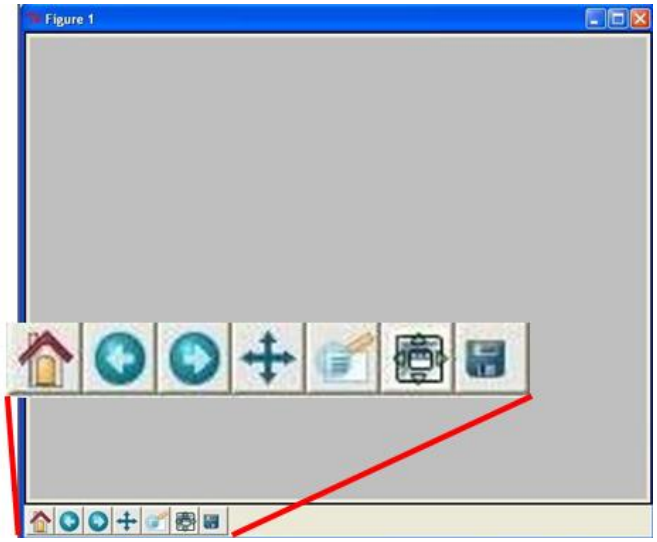
Introduzione a Pylab

L'interfaccia Pylab costituisce il modo più semplice per lavorare con Matplotlib.

Le funzioni sono molto simili all'ambiente Matlab.

Esempio

```
>>>from pylab import *
>>>figure()
>>>show()
```

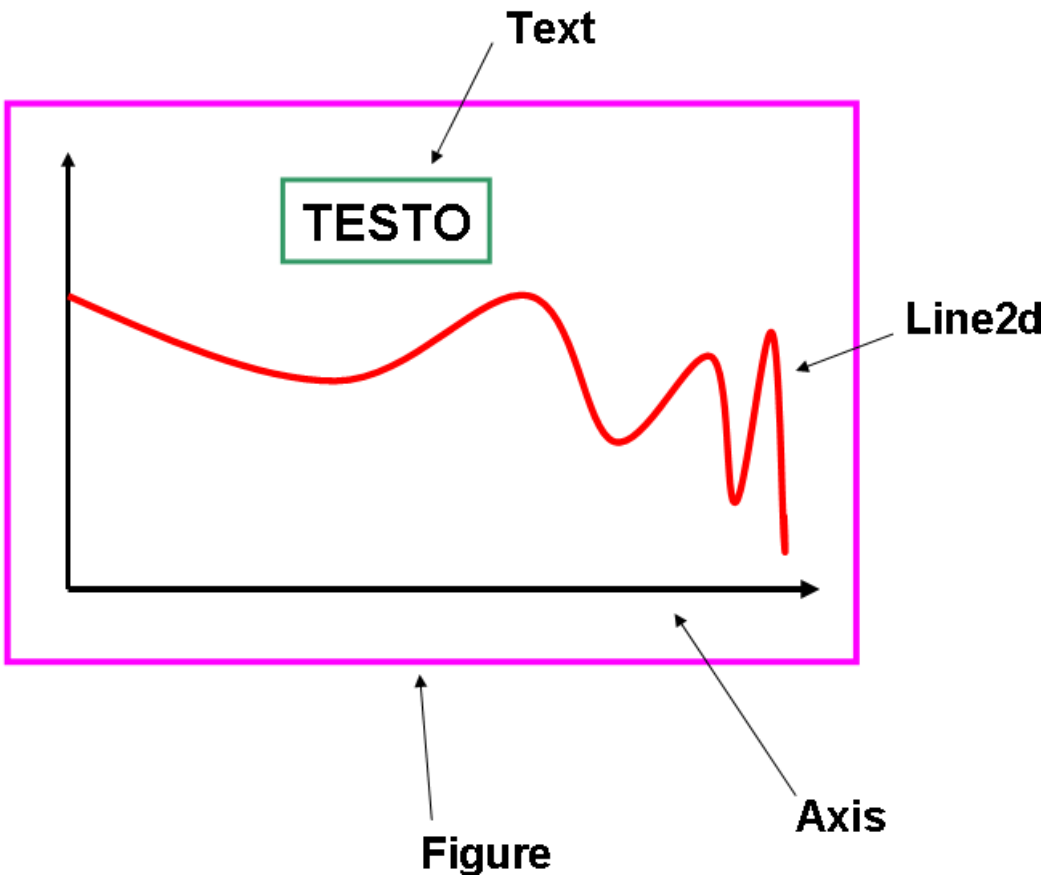


La funzione *figure()* istanzia un oggetto figura.

La funzione *close(n)* chiude la finestra *n*

La funzione *show()* visualizza tutte le figure create

Introduzione a Pylab



Le principali entità su cui lavorare sono:

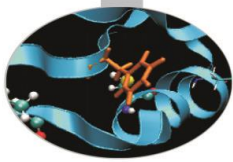
Figure l'oggetto figure ha attributi propri (risoluzione,dimensioni,).

Line2d le linee2d possiedono diverse proprimarcatori, etc.

Text è possibile modificare e gestire testo (plain o math)

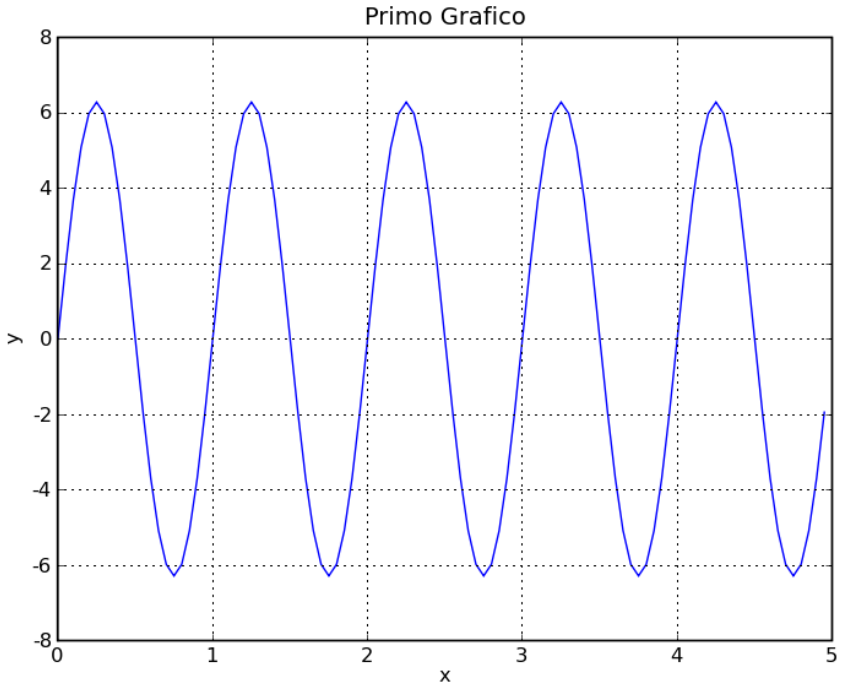
Axis per la gestione degli assi

Comandi di base



Esempio: Primo grafico in Pylab

```
>>>from numpy import *
>>>from pylab import *
>>>t=arange(0,5,0.05)
>>>f=2*pi*sin(2*pi*t)
>>>plot(t,f)
>>>grid()
>>>xlabel('x')
>>>ylabel('y')
>>>title('Primo grafico')
>>>show()
```



NOTA

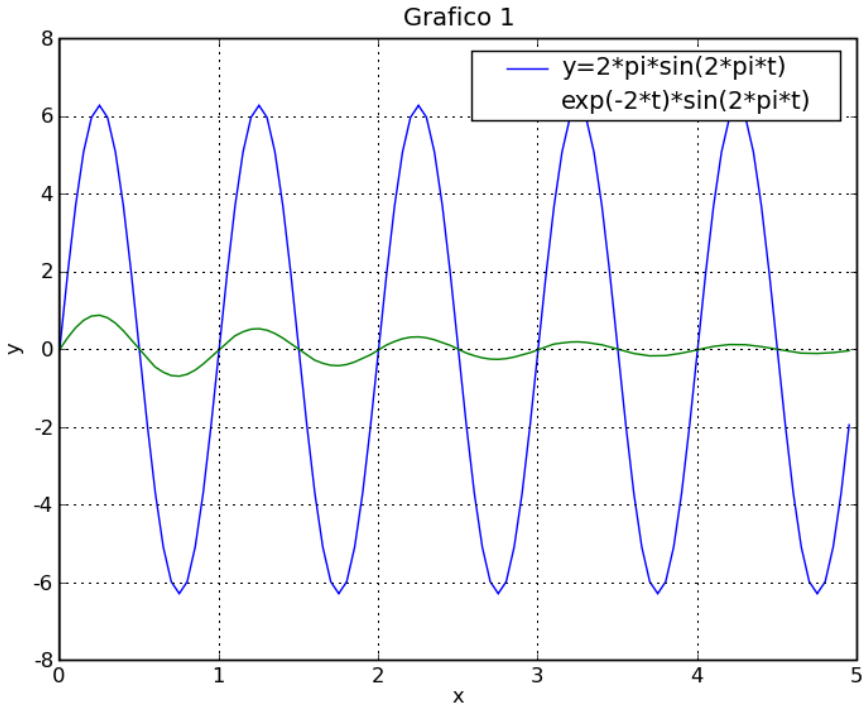
Il grafico viene visualizzato solo alla chiamata della funzione show().

Per lavorare interattivamente è necessario impostare:

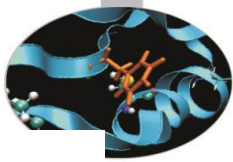
- mode interactive — *rcParams['interactive']=True*
- il tipo di backend — *rcParams['backend']='TkAgg'*

Comandi di base

```
>>>hold(True)
>>>f2=sin(2*pi*t)*exp(-2*t)
>>>plot(t,f2)
>>>legend(('y=2*pi*sin(2*pi*x)','sin(2*pi*x)*exp(-2*x)'))
```



Comandi di base

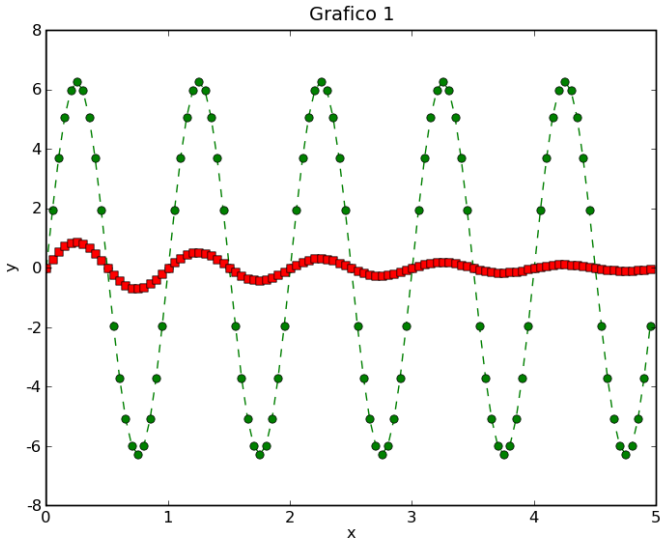


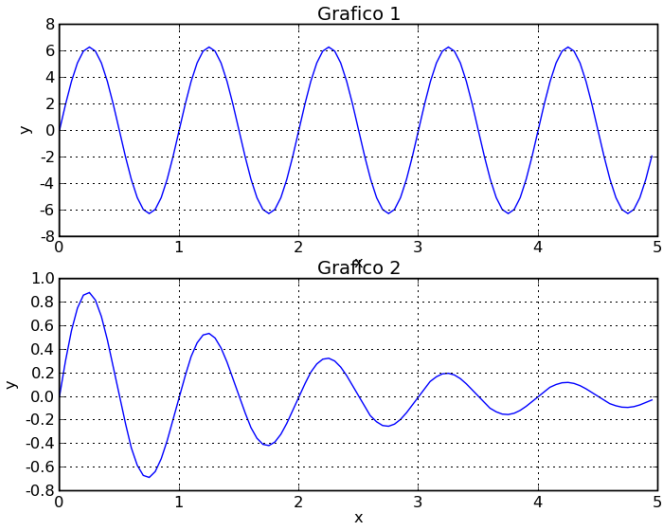
In alternativa :

```
>>>clf
>>>plot(t,f,'g--o',t,f2,'r:s')
>>>xlabel('x')
>>>ylabel('y')
>>>title('Grafico 1')
```

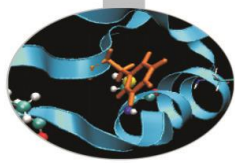
SUBPLOT

```
>>>subplot(211)
>>>plot(t,f)
>>>xlabel('x');ylabel('y') ; title('Grafico 1')
>>>subplot(121)
>>>plot(t,f2)
>>>xlabel('x');ylabel('y') ; title('Grafico 2')
```





Figure



E’ possibile gestire e creare un numero arbitrario di figure tramite il comando *figure().*

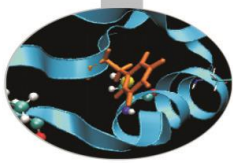
E’ possibile gestire i seguenti attributi della figure:

- *figsize* dimensione in inches
- *facecolor* colore di riempimento
- *edgecolor* colore del bordo
- *dpi* risoluzione
- *frameon* per mantenere il background grigio alla figura.

Per chiudere la figura si possono usare i comandi:

- *close(num)*
- *close(istance)*
- *close(‘all’)*

Plot e Subplot



Il comando

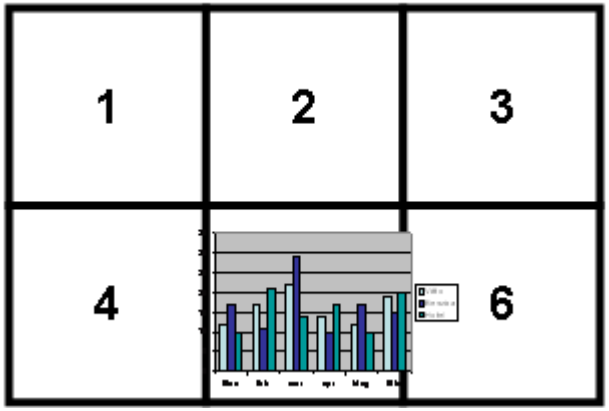
plot(line2d , [properties line2d])

è un comando versatile che consente di creare grafici multilinea specificando lo stile.

Il comando

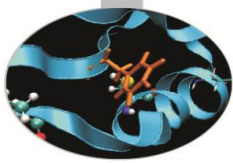
subplot(nrows,ncol,index)

Permette di creare grafici multipli su una griglia con un numero specifico di righe e di colonne.



subplot(2,3,5)

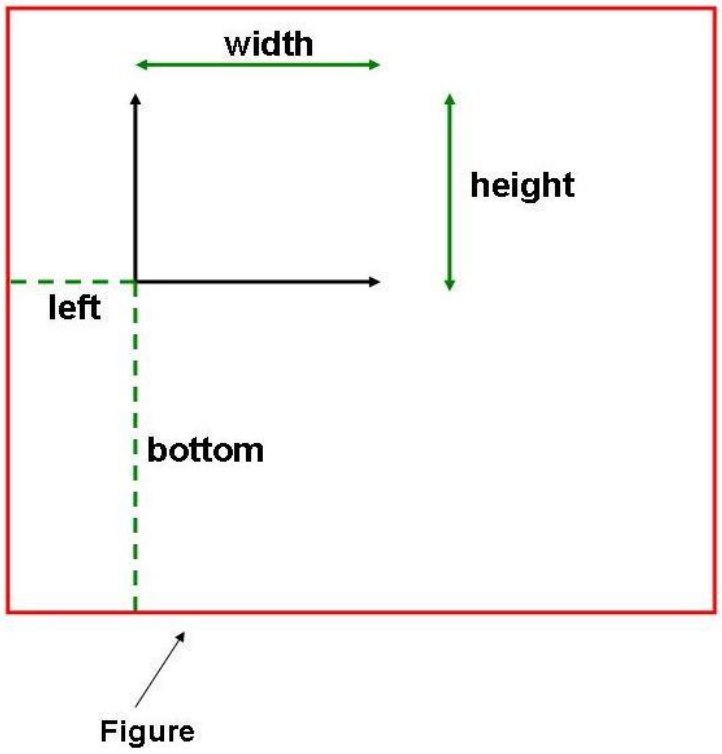
Axes



L'oggetto *axes()* permette la gestione degli assi e si comporta in maniera simile a subplot.

axes() equivale a subplot(111)

axes([left,bottom, width, height]) posiziona e dimensiona il grafico secondo la lista di parametri passati come argomento.



Alcuni metodi

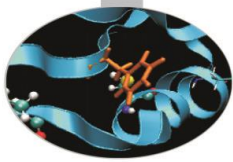
axis([*xmin,xmax,ymin,ymax*])

grid()

xticks(*location,label*)

legend([*list_lines*],[*list_label*], *loc*, [*text_prop*])

Line2D Properties



L'oggetto linea ha diversi attributi: è possibile modificare le dimensioni, lo stile, il colore etc. La funzione:

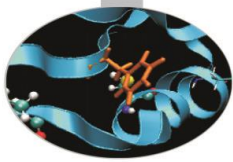
*setp(*args, **kwargs)*

permette di cambiare tali attributi.

In alternativa è possibile modificare gli attributi tramite i metodi dell'oggetto line2D.

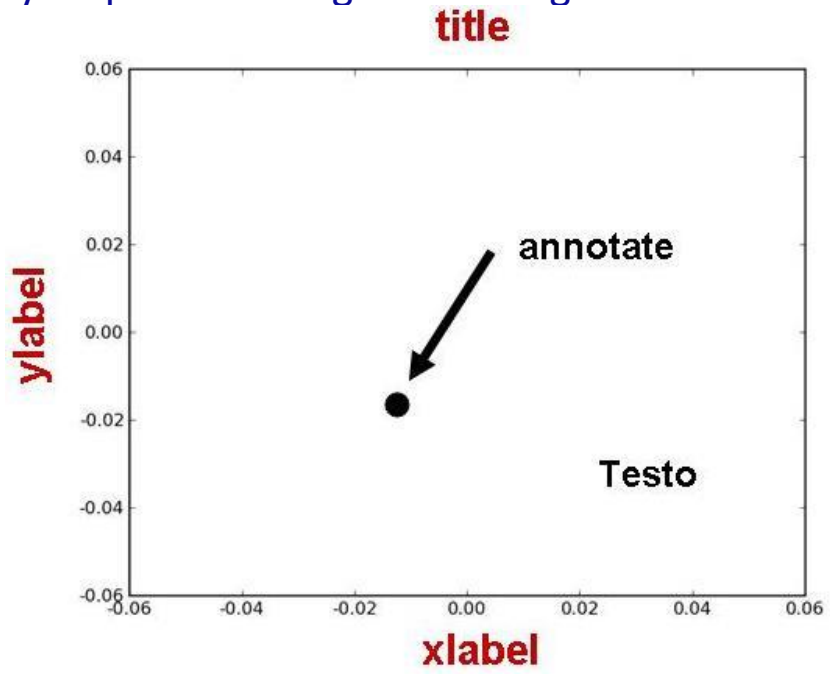
Tra gli attributi ricordiamo:

- *color* 'b', 'r', 'g', 'y', ' k', 'w , 'c', 'm'
- *linewidth* float
- *linestyle* '', '-', '- -', ':', '.-'
- *label* stringa
- *marker* '.', 'o', 'D', '^', 's', '*', '+', 'h'
- *markersize float*
- *markerfacecolor* color



Gestione del testo

Pylab permette di gestire stringhe di testo all'interno di grafici.



xlabel (s, *args, **kwargs)

ylabel (s, *args, **kwargs)

title (s, *args, **kwargs)

annotate(s, xy, xytext=None, xycoords='data', textcoords='data',

arrowprops=None,**props)

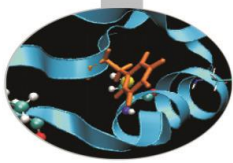
text(x, y, s, fontdict=None, **kwargs)

Inoltre Pylab è in grado di inglobare espressioni matematiche in espressioni di testo utilizzando la sintassi LaTex. Per esempio la sintassi:

xlabel(r'$y_i=2\pi \sin(2\pi x)$') equivale a $y_i = 2\pi \sin(2\pi x)$

E' necessario inoltre imporre: rcParams(text.usetex)=True

Text Properties



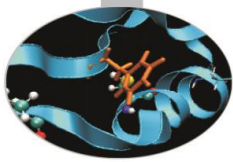
L'oggetto testo possiede le seguenti proprietà:

- *Fontsize* xx-small, x-small, small, medium, large, x-large, xx-large
- *Fontstyle* *normal, italic, oblique*
- *Color*
- *Rotation* *degree , 'vertical', 'horizontal'*
- *Verticalalignment* *'top', 'center', 'bottom'*
- *Horizontalalagnment* *'left', 'center', right'*

Gli attributi possono essere modificati in tre modi:

- Tramite *keyword arguments,* tramite la funzione *setp,* tramite i metodi dell'oggetto testo:

```
>>>xlabel('ciao', color ='r', fontsize='large')        #keyword arguments
>>>l=ylabel('asse y')
>>>setp(l,rotation=45)                                  #setp()
>>>l.set_color('r')                                    #object method
```



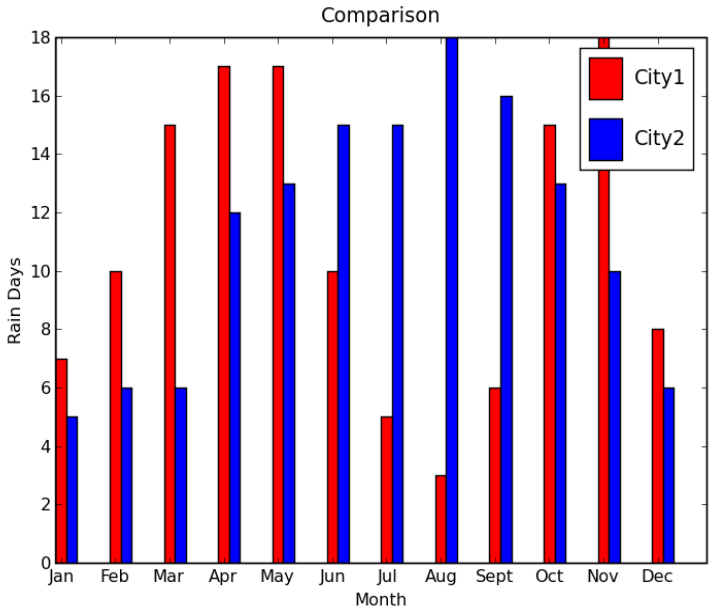
Diagrammi a barre

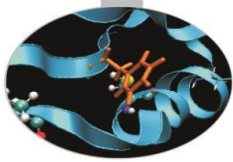
Come creare un diagramma a barre:

bar(left, height)

Esempio:

```
from pylab import *
n_day1=[7,10,15,17,17,10,5,3,6,15,18,8]
n_day2=[5,6,6,12,13,15,15,18,16,13,10,6]
m=['Jan','Feb','Mar','Apr','May','Jun'
,'Jul','Aug','Sept','Oct','Nov','Dec']
width=0.2
i=arange(len(n_day1))
r1=bar(i, n_day1,width, color='r',linewidth=1)
r2=bar(i+width,n_day2,width,color='b',linewidth=1)
xticks(i+width/2,m)
xlabel('Month'); ylabel('Rain Days'); title('Comparison')
legend((r1[0],r2[0]),('City1','City2'),loc=0,labelsep=0.06)
```



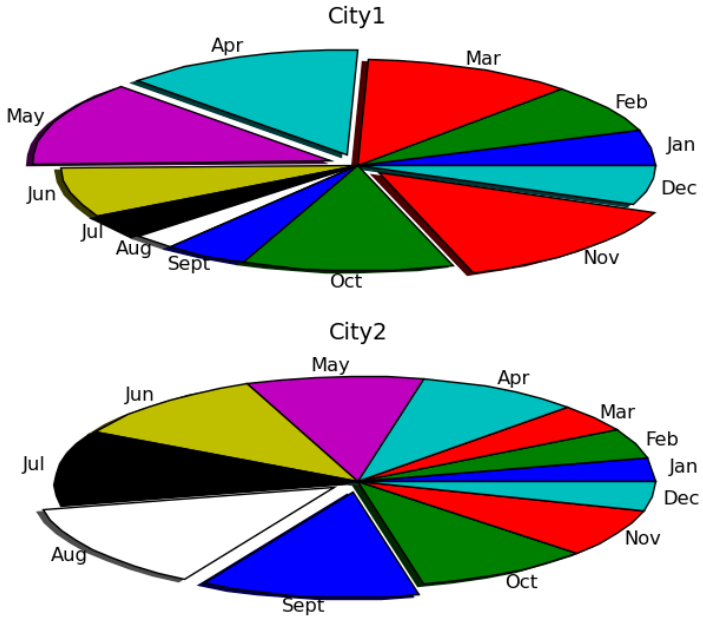


Torta

Oppure con gli stessi dati come creare una torta:

pie(x)

```
subplot(211)
pie(n_day1,labels=m,
explode=[0,0,0,0.1,0.1,0,0,0,0,0,0.1,0],
shadow=True)
title('City1')
subplot(212)
pie(n_day2,labels=m,
explode=[0,0,0,0,0,0,0,0.1,0.1,0,0,0],
shadow=True)
title('City2')
```

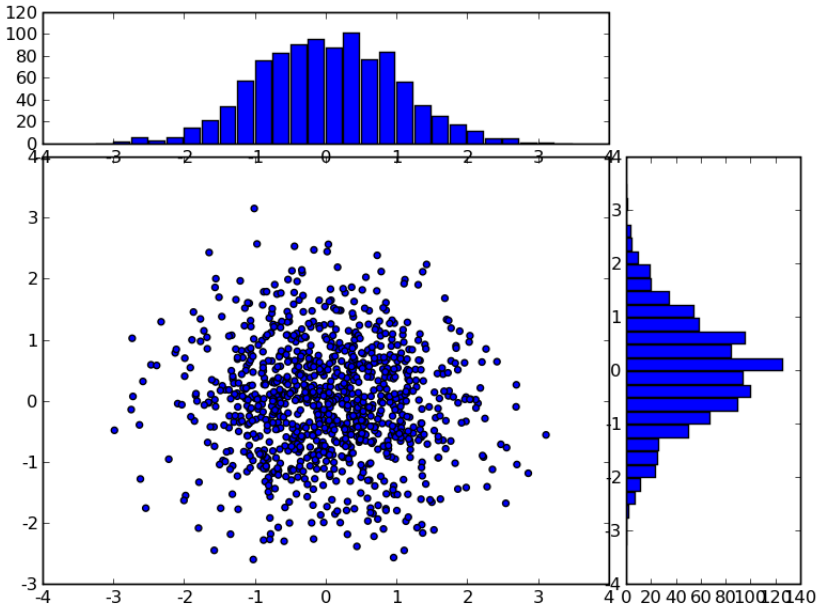


Scatter plot - Istogrammi

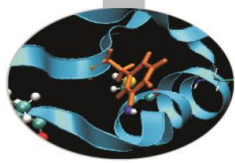
scatter(x,y)

hist(x)

```
x = numpy.random.randn(1000)
y = numpy.random.randn(1000)
axscatter = axes([0.1,0.1,0.65,0.65])
axhistx = axes([0.1,0.77,0.65,0.2])
axhisty = axes([0.77,0.1,0.2,0.65])

axscatter.scatter(x, y)
binwidth = 0.25
xymax = max( [max(fabs(x)), max(fabs(y))] )
lim = ( int(xymax/binwidth) + 1) * binwidth
bins = arange(-lim, lim + binwidth, binwidth)
axhistx.hist(x, bins=bins)
axhisty.hist(y, bins=bins, orientation='horizontal')
show()
```



Meshgrid

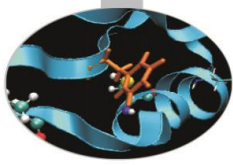


- Come costruire una griglia bidimensionale?
- Data una griglia (xi,yi) vogliamo calcolare per ciascun punto della griglia il valore della funzione f(xi,yi)

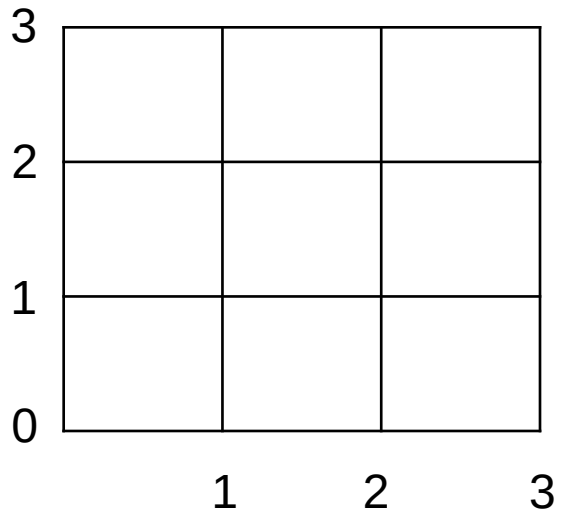
```
>>> x=np.arange(4)
>>> y=np.arange(4)
>>> def f(x,y):
        return x**2+y
>>> x
array([0, 1, 2, 3])
>>> y
array([0, 1, 2, 3])
>>> f(x,y)
array([ 0,  2,  6, 12])
```

WRONG!!

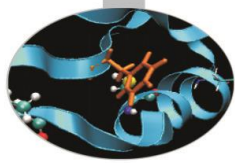
Meshgrid



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



OK!!



Contour plot

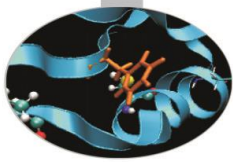
contourf(*args, **kwargs)

contour(*args,**kwargs)

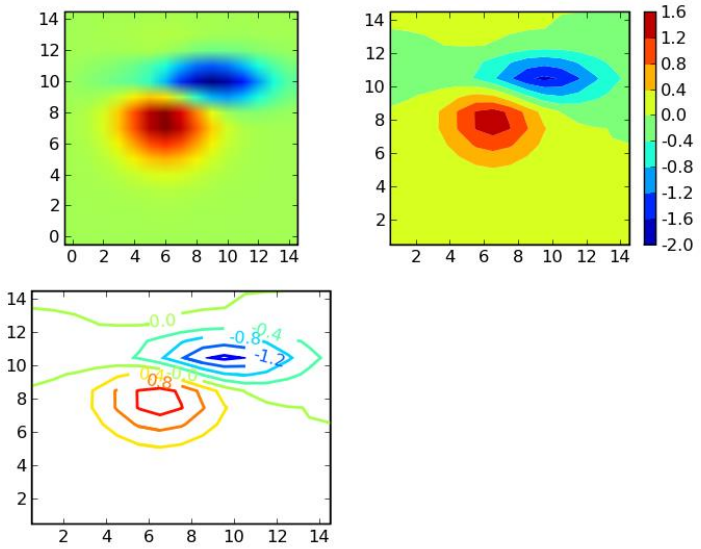
meshgrid(x,y)

```
from pylab import *
delta = 0.5
x = arange(-3.0, 4.001, delta)
y = arange(-4.0, 3.001, delta)
X, Y = meshgrid(x, y)
Z1 = bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = (Z1 - Z2) * 10
levels = arange(-2.0, 1.601, 0.4)

figure()
subplot(221)
imshow(Z,origin='lower')
```

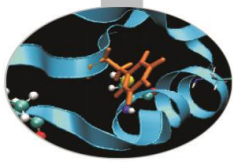
Contour plot



```
subplot(222)
l= contourf(Z,levels,origin='lower')
colorbar(l)
subplot(223)
l= contour(Z, levels,origin='lower',linewidths=2)
clabel(l,inline=1, fmt='%1.1f',fonsize=14)
show()
```



Output



Matplotlib supporta diversi backend grafici. Possiamo dividere la tipologia di backend in due categorie:

- User interface backend: per l'assemblaggio in GUI. In Python esistono diverse librerie per la costruzione di interfaccie grafiche tra cui Tkinter, PyQt, pygtk che vengono supportate da matplotlib.
- Hardcopy backend: per la stampa su file. Vengono supportati i seguenti formati *.jpg, *png, *svg, *pdf, *rgba.