Python Lecture 4 – Libraries

- Numpy
- Scipy
- Matplotlib

Bibliography and learning materials

★ Bibliography:

https://docs.scipy.org/doc/

http://docs.python.it/

https://matplotlib.org/

and much more available in internet

★ Learning Materials:

https://github.com/bertocco/bash_lectures

Numpy

numpy states for Numerical Python.

NumPy is the fundamental package for scientific computing in Python.

NumPy is a Python library that provides:

- ★ a multidimensional array object,
- ★ various derived objects (such as masked arrays and matrices),
- ★ an assortment of routines for fast operations on arrays, including:
  - mathematical, logical, shape manipulation
  - sorting
  - selecting
  - I/O
  - discrete Fourier transforms
  - basic linear algebra
  - basic statistical operations
  - random simulation
  - and much more…..

Numpy module organization

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

Scipy

SciPy is a collection of

- mathematical algorithms and
- convenience functions

built on the numpy extension of Python.

It provides the user with high-level commands and classes for manipulating and visualizing data.

Using an interactive Python session with scipy we have a data-processing and system-prototyping environment rivaling systems such as MATLAB and IDL.

Scipy modules

SciPy is organized into subpackages covering different scientific computing domains:

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distribution and function

Scipy sub-packages need to be imported separately.

Example:

```
from scipy import linalg, io
```

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code.

For simple plotting the pyplot sub-module provides a MATLAB-like interface, particularly when combined with IPython. It provides users with full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

How to find documentation (1)

- The dir(module) function can be used o look at the namespace of a module or package, i.e. to find out names that are defined inside the module.

- The help(function) function is available for each module/object and allows to know the documentation for each module or function.

- Try (in the interpreter) the commands:
  import math
  dir()
  help(math.acos)

- The type(object) function allows to know the type of the object passed as argument.
  l = [1, "alfa", 0.9, (1, 2, 3)]; print [type(i) for i in l]

★The source(function) function, when given a function written in Python as an argument, prints out a listing of the source code for that function. This can be helpful in learning about an algorithm or understanding exactly what a function is doing with its arguments.

How to find documentation (2)

numpy/scipy-specific help system is also available under the command numpy.info.

Example (try):

```
>>> import scipy.optimize
>>> import numpy as np
>>> np.info(scipy.optimize.fmin)
```

If you use a second keyword argument of numpy.info, it defines the maximum width of the line for printing. If a module is passed as the argument to help then a list of the functions and classes defined in that module is printed.

Example (try):

```
>>> np.info(scipy.optimize)
```

Name convention

Generally, for brevity and convenience, it is used a convention on names used to import packages (numpy, scipy, and matplotlib):

```
>>> import numpy as np
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
```

Generally scipy is not imported as module because interesting functions in scipy are actually located in the submodules, so submodules or single functions are imported:

NOT used	**used**
`import scipy`	`from scipy import fftpack` `from scipy import integrate`

The scipy namespace itself only contains functions imported from numpy. Therefore, importing only the scipy base package does only provide numpy content, which could be imported from numpy directly.

These functions still exist for backwards compatibility, but should be imported from numpy directly.

numpy

Python arrays: numpy ndarray

ndarray object is an n-dimensional array of homogeneous data types, with many operations being performed in compiled code for performance.

Important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

To know how to use NumPy arrays is needed to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software because a growing plethora of scientific and mathematical Python-based packages are using NumPy arrays.

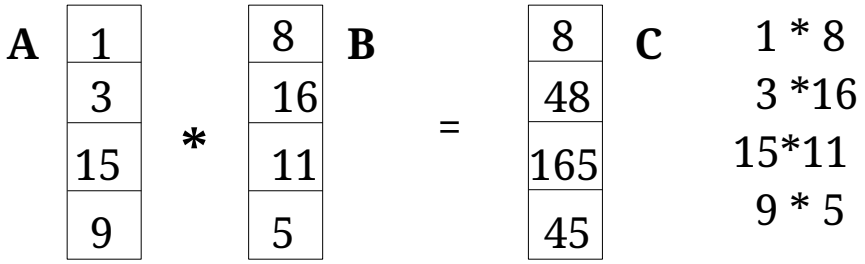
sequences vs ndarray by examples (2)

It is important to understand differences and performance issues related to the use of python built-in sequences and numpy.ndarray

We are going to learn by examples.

As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```



This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the **inefficiencies** of looping in Python.

We could accomplish the same task much more quickly in C by writing (variable declarations and initializations, memory allocation, etc. are neglected)

```
for (int i = 0; i < rows; i++) {
  c[i] = a[i]*b[i];
}
```

This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python

sequences vs ndarray by examples (3)

If the array dimension increases, the coding work increases also and the produced code is less readable.

C code example with a 2-D array:

```
for (int i = 0; i < rows; i++) {
  for (int j = 0; j < columns; j++) {
    c[i][j] = a[i][j]*b[i][j];
  }
}
```

sequences vs ndarray by examples (4)

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when an ndarray is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

```
c = a * b
```

does what the earlier examples do, at near-C speeds, but with the code simplicity we expect from something based on Python. Indeed, the NumPy idiom is even simpler!

Vectorization and broadcasting

This last example illustrates two of NumPy's features which are the basis of much of its power: vectorization and broadcasting.

Broadcasting is the term used to describe the implicit element-by-element behavior of operations.

In NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion.

In the example above, a and b could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays of with different shapes, provided that the smaller array is "expandable" to the shape of the larger in such a way that the resulting broadcast is unambiguous.

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just "behind the scenes" in optimized, pre-compiled C code. Main vectorized code advantages are:

- vectorized code is more concise and easier to read
- fewer lines of code generally means fewer bugs
- the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)

numpy array glossary (1)

array size is the number of elements in the array

array rank is the number of axis/dimensions of the array

array shape is the array dimension, i.e. an integer tuple containing the number of integers for each dimension

The shape attribute specifies the array shape. **Example**:

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

- ndim specifies the array dimension

```
b.ndim
3
```

numpy array glossary (2)

itemsize allows to specify the dimension of each array element.

```
>>>b = np.array([[1,2,3],[4,5,6]])
>>> b.itemsize
8
>>> b.dtype
dtype('int64')
>>> b.strides              # bytes to jump to get to the next element
                           # of each dimension
(24, 8)                    # (skype_byte_row, skype_byte_col)

>>>c = np.array([[1,2,3],[4,5,6]], order='F')   # Fortran order
>>> c.strides             # tuple of bytes to step in each dimension when
                           # traversing an array
(8, 16)
```

array memory allocation

Memory allocation refers to data store.

- C-style memory allocation stores multi-dimensional data in row-major order in memory
- Fortran-style memory allocation stores multi-dimensional data in column-major order in memory

Array to store:

1	2	3
4	5	6

1	4	2	5	3	6

Fortran - Style

1	2	3	4	5	6

C-Style

array creation

The easiest way to create an array is the function:

array(object, dtype=None, copy=1,order=None) -> array

allowing to convert the ‘object’ structure in an array.

Example:

```
>>>import numpy as np
>>>a = np.array([1,2,3,4])
>>>list = [1,2,3,4]
>>>tupla = (5,6,7,8)
>>>a = np.array(list)              # from a list
>>>b = np.array(tupla)            # from a tuple
>>>c = np.array([list,tupla])     # from a list and from a tuple
>>>a.dtype                        # check the array type
dtype('int64')
```

dtype

A numpy array is homogeneous, and contains elements described by a dtype object.

A dtype object can be constructed from different combinations of fundamental numeric types.

Numpy supports more data types than pure python (see table).

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

dtype definition and usage

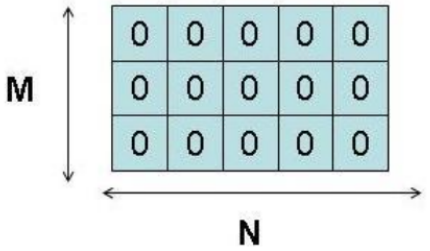
Array with new user defined types (structure) can be created:

```
>>>dt = np.dtype([('Name','S3'),('Anni', np.int64)])
>>a = np.array([('Chiara',3),('Marco',4)],dtype=dt)
>>> a
array([(b'Chi', 3), (b'Mar', 4)], dtype=[('Name', 'S3'), ('Anni', '<i8')])

>>>dt = np.dtype({'names':('Name','Anni','Cognome'),'formats':('S3','int64','S3')})
>>>a = np.array([('Chiara',3,'Bianchi'),('Marco',4,'Rossi')],dtype=dt)
>>> a
array([('Chiara', 3L, 'Bia'), ('Marco', 4L, 'Ros')],
dtype=[('Name', '|S10'), ('Anni', '<i8'), ('Cognome', '|S10')])
```

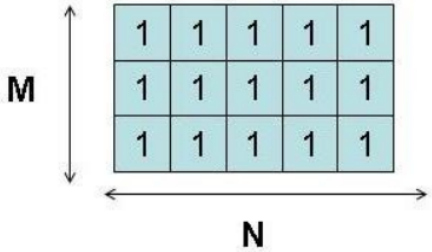
Other array creations

If the array content is unknown, there are functions to fill the array.

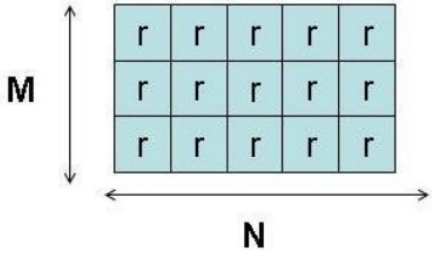
- zeros(shape, dtype=float, order =‘C’) function
  create an array of 0 of shape dimension



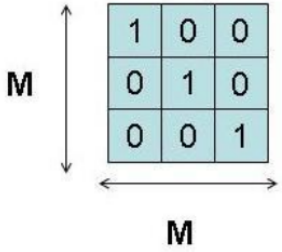
- ones(shape,dtype=None, order =‘C’)
  create an array of 1 of shape dimension



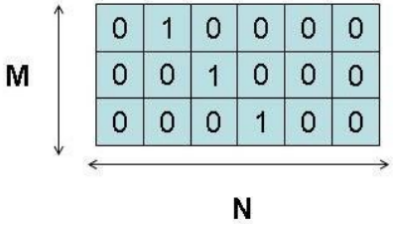
- empty(shape, dtype=None, order =‘C’)
  creates an array with shape dimension without initializing it



- identity(n, dtype=None)
  creates the NxN identity matrix



- eye(N, M=None, k=0, dtype=float)
  creates an MxM matrix filling with 1 the k-esima diagonal



Note: order : {‘C’, ‘F’}, optional, default: ‘C’. Means whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

arange() and linspace()

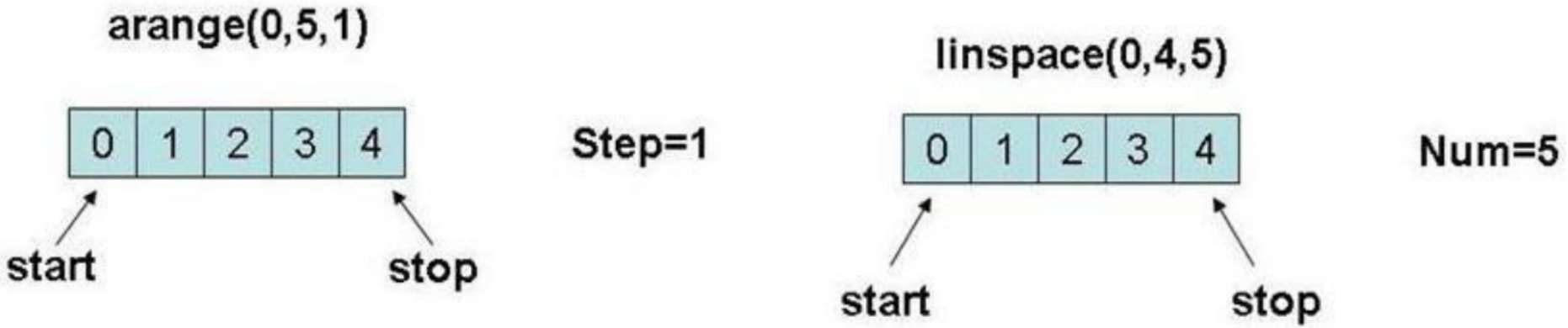
An array can be created from a numbers sequence with functions similar to function range() for lists:

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

creates an array of numbers between '*start*' and '*stop*' with step '*step*'

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

creates a sequence of *num* numbers uniform distributed between *start* and *stop*



Create an array from string

An array can be created from a string using the function fromstring()

Example:

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

Numerical operations on arrays

Numerical operators in numpy acts elementwise (element-by-element) on arrays.

This rule is valid both for unary and binary operators and also for transcendental functions (like sin, cos, log, etc.)

Example:

```
b=np.array([5,6,7,8])
c=np.arange(1,5)
d=c+b
print (“Sum “ ,b,”+”,c, “= “, b+c)
b+=1
print (“Increment b +=1 b=“, b)
print (“Multiply c*3 ” ,c, “* 3= ”,c*3)
print (“Sin (c)”, np.sin(c))
```

To deep:

http://scipy-lectures.org/intro/numpy/operations.html

Output:

```
Sum [5,6,7,8] + [1,2,3,4] = [6,8,10,12]
Increment b+=1 b= [6,7,8,9]
Multiply c*3 [1,2,3,4] *3 = [3,6,9,12]
Sin(c) [ 0.84147098, 0.90929743, 0.14112001, -0.7568025 ]
```

Numerical operations: comparisons

Comparisons:

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

Array-wise comparisons:

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

Numerical operations: logical ops and trascendental functions

Logical operations:

```
>>> 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)
```

Transcendental functions:

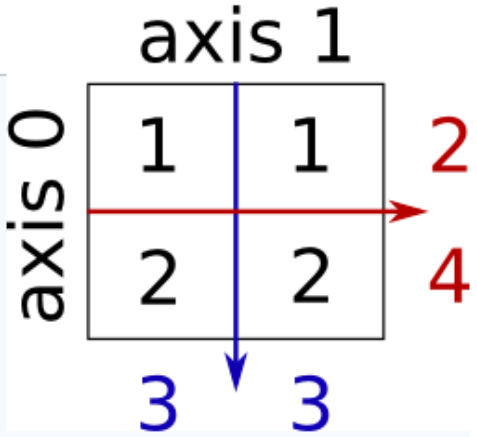
```
>>> a = np.arange(5)
>>> np.sin(a)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
>>> np.log(a)
array([       -inf,  0.        ,  0.69314718,  1.09861229,  1.38629436])
>>> np.exp(a)
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,  54.59815003])
```

Computing sums

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sum(x)
10
>>> x.sum()
10
```

Sum by rows and by columns:

```
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>> x.sum(axis=0)   # columns (first dimension)
array([3, 3])
>>> x[:, 0].sum(), x[:, 1].sum()
(3, 3)
>>> x.sum(axis=1)   # rows (second dimension)
array([2, 4])
>>> x[0, :].sum(), x[1, :].sum()
(2, 4)
```



Reshaping and resizing arrays

Methods resize e reshape allow to modify shape and dimension of an array.

- reshape(shape, order='C')

  Return a new data structure with array elements re-distributed on the base of the new shape with the new order

  With reshape() the number of array elements is unmodified

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

  Allow to modify the array shape and the dimension also

  Resize gives an error if the array is referenced.

Examples:

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

```
>>>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
```

reshape() example (try)

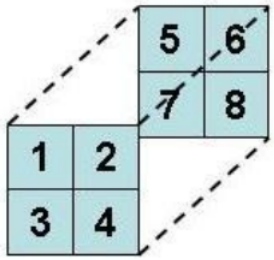
```
>>> a=array(range(1,9))

>>> 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
```

1	2	3	4	5	6	7	8

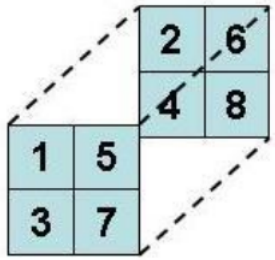
reshape() example output

Shape (8,)

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

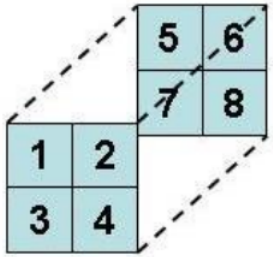


C - Style

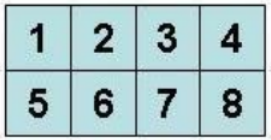


Fortran - Style

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



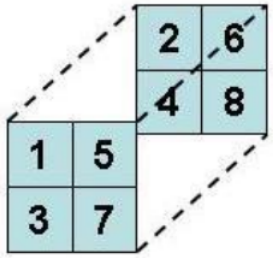
C - Style



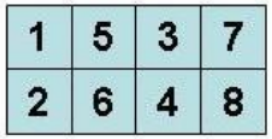
Reshape

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

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



Fortran - Style



Reshape

indexing – slicing – iteration (1)

Array Indexing e Slicing

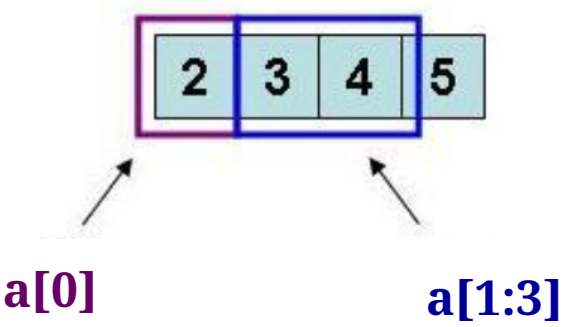
Access to the elements of an array is achieved using the [] operator.

The slicing operator [:] is also applicable to arrays.

In the case of one-dimensional arrays, the same notation as for lists is adopted.

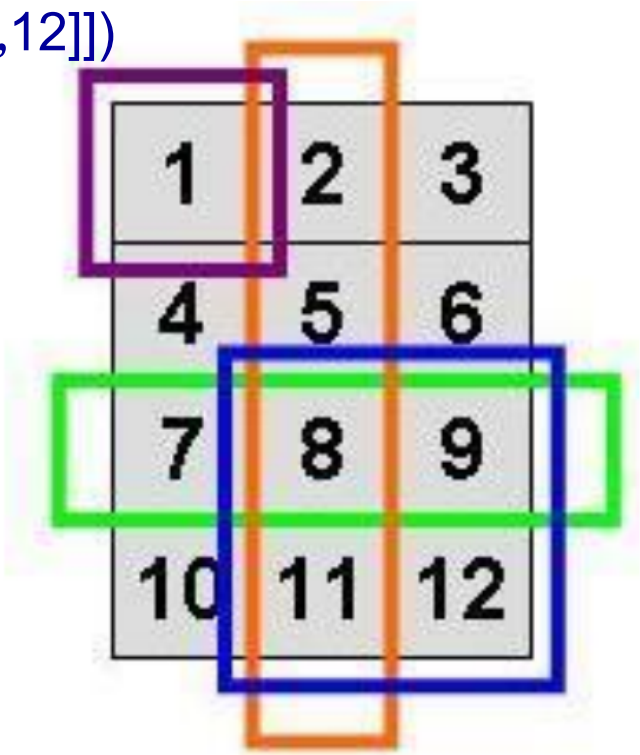
Esempio

```
>>> a = np.ones(4)
>>> a
array([ 1.,  1.,  1.,  1.])
>>> b = np.arange(1,5)
>>> b
array([1, 2, 3, 4])
>>> a+=b ;  a          # a+=b means a=a+b
array([ 2.,  3.,  4.,  5.])
>>> print (“a[0] “, a[0])
>>> 2.0
>>> a[1:3] = a[1:3]*3     # Modify the elements from 1 to 3
>>> print (a)
>>> [ 2., 9., 12., 5.]
```



indexing – slicing – iteration (2)

```
>>>a=array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
>>> 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 (3)

Example:

```
>>> a=np.arange(25)
>>> a=a.reshape((5,5)) ; print(a)
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]])
>>> print(a[::,1])
array([ 1, 6, 11, 16, 21])
>>> print(a[1])
array([5, 6, 7, 8, 9])
>>> print(a[1,:])
array([5, 6, 7, 8, 9])
>>> print(a[1,::])
array([5, 6, 7, 8, 9])
>>> print(a[1,::2])
array([5, 7, 9])
>>> print(a[1,10::-1])
array([9, 8, 7, 6, 5])
```

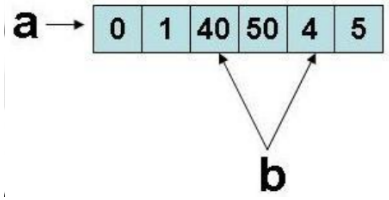
indexing – slicing – iteration (4)

Note:

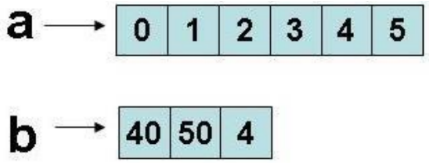
The slicing operation for numpy arrays is different from slicing for python built-in lists:

- in numpy array slicing the generated sub-array is a reference to the original memory area (shallow copy)
- in built-in python lists the generated sub-list is a by-value copy of the original memory area

```
>>> a=np.arange(6) ; a
array([0, 1, 2, 3, 4, 5])
>>> b=a[2:5] ; b
array([2, 3, 4])
>>> 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) ; a
[0, 1, 2, 3, 4, 5]
>>> b=a[2:5] ; b
[2, 3, 4]
>>> b[0]=40
>>> b[1]=50
>>> print ("a:", a , "b:", b)
a: [ 0 1 2 3 4 5] b: [40 50 4]
```



This impacts on performances and memory consumption and results.

indexing – slicing – iteration (5)

The copy is done element-by-element and the two objects are different.

Example:

```
>>> a = np.arange(5)
>>> b = np.zeros_like(a)   # Return an array of zeros with the same shape and type as a given array.
>>> b[:] = a[:]            # Copy is element-by-element and the two objects are different
                           # (deep copy)
>>> b[3] = 1000
>>> b == a
array([True,True,True,False,True],dtype=bool)
```

indexing – slicing – iteration (6)

Iteration on array elements can be done in different ways

- by default iteration is along the first axis

  Example:

  ```
  >>>a=np.arange(9)
  >>> for el in a:
  …       print (el)
  >>> 0 1 2 ….
  ```

- by for along one defined axis

  Example:

  ```
  >>> a=np.arange(9)
  >>>a.shape=(3,3)
  >>>for i in arange(a.shape[0]):
  …       for j in arange(a.shape[1]):
  …             a[i,j] = (i + j)
  ```

- Through flat iterator on every array element.

  Esempio:

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

Note:
The array iteration is computationally no efficient.

Python has specific functions (C written) to work directly on the entire array (more efficient).
Avoid *for loop* when possible.

Array selection

The selection of array elements can be done in other different, more complex ways :

– Through an array of indices

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

The new array:

➔ has the same shape of the array of indices

➔ has the same type and values of the original array

– Using a boolean mask

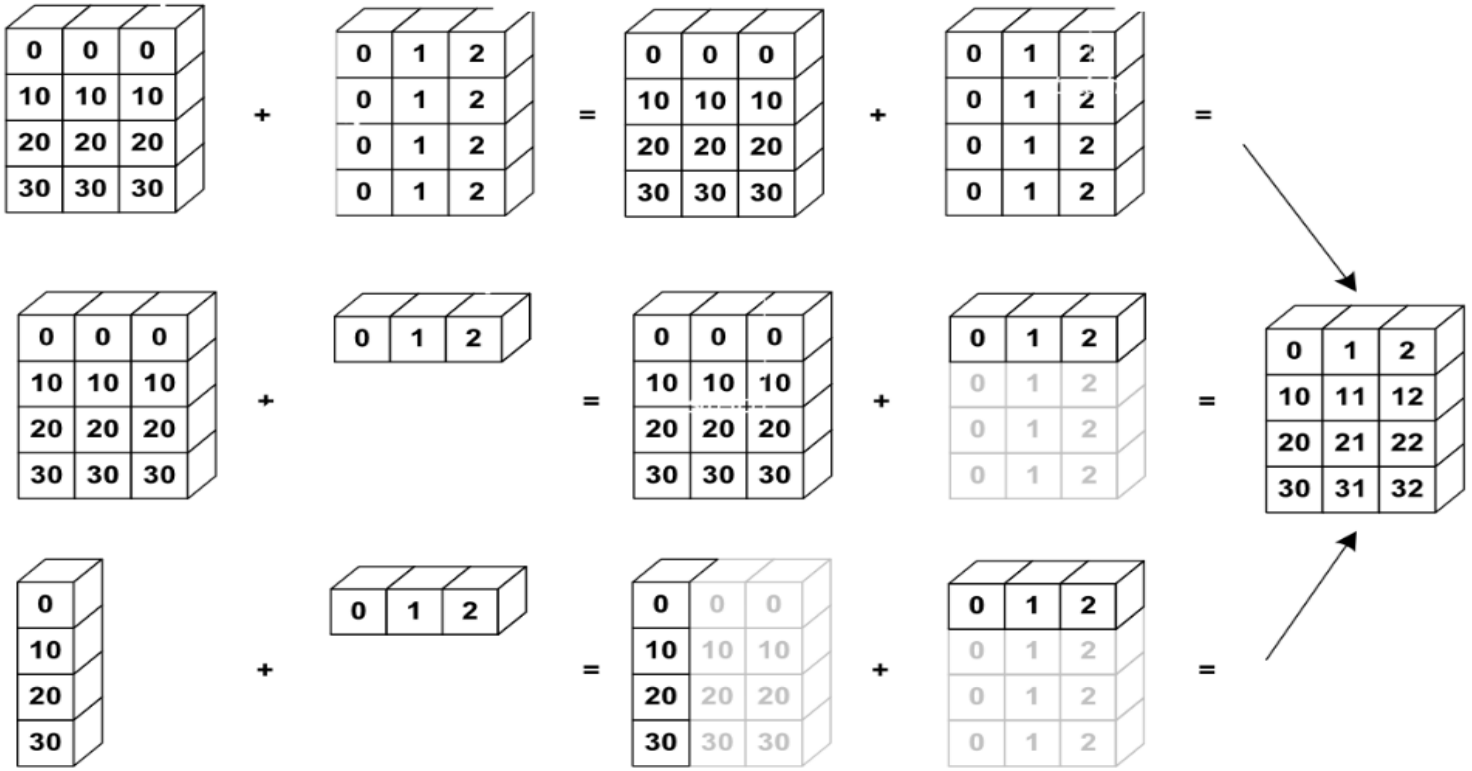
```
>>> y = np.arange(18); y
>>> y=y.reshape(3,6); y
array([[ 0, 1, 2, 3, 4, 5],
[ 6, 7, 8, 9, 10, 11],
[12, 13, 14, 15, 16, 17]])
>>> y[y>10]
array([11, 12, 13, 14, 15, 16, 17]          # 1d array
```

Broadcasting

Basic operations on numpy arrays (addition, etc.) are elementwise (element-by-element)

This works on arrays of the same size.

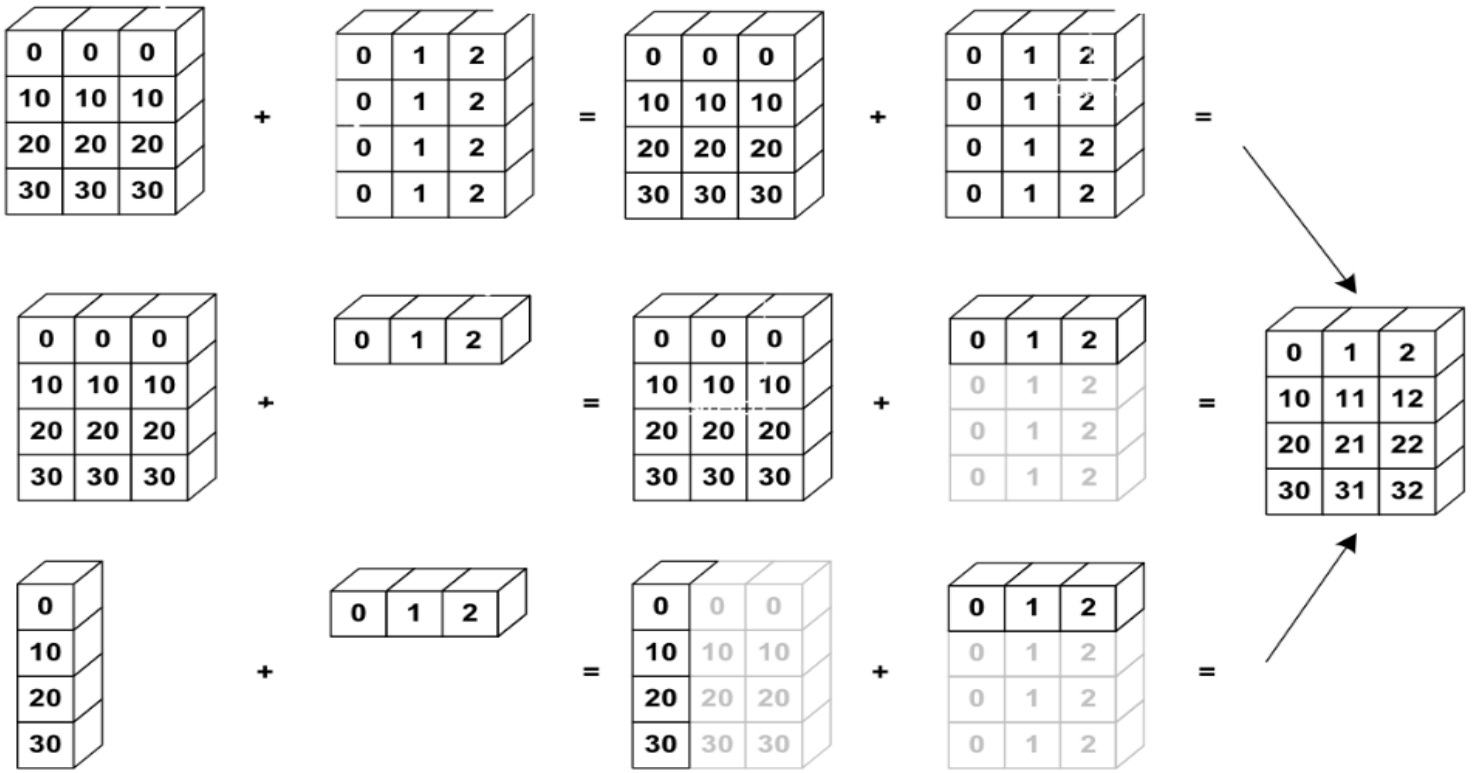
Nevertheless, It's also possible to do operations on arrays of different sizes if NumPy can transform these arrays so that they all have the same size: this conversion is called broadcasting.
The image below gives an example of broadcasting:



Broadcasting rules

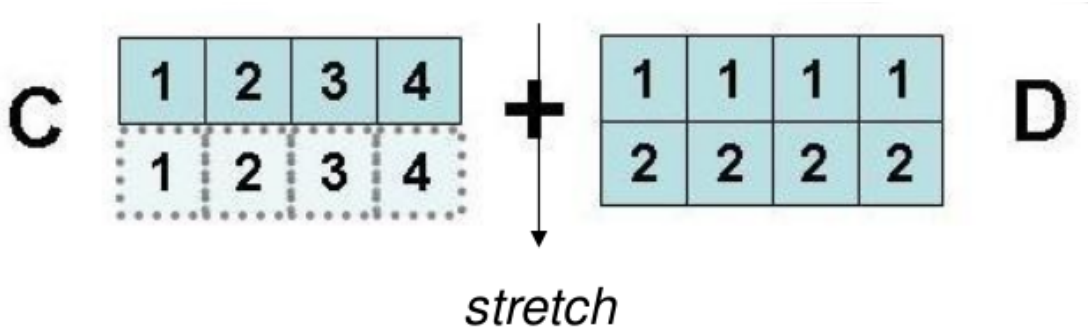
The broadcasting has two rules:

- If the two arrays have not the same number of dimension
  then the more little array is re-shaped (adding dimension ‘1’ until both arrays have the same dimension
- Arrays with dimension ‘1’ along one direction behaves as the array bigger along that version. The value is repeated along the broadcast direction.



Broadcasting example

```
c=np.arange(1,5)
d=np.array([[1,1,1,1],[2,2,2,2]])
print d, ”+”, c “= “ d+c
```

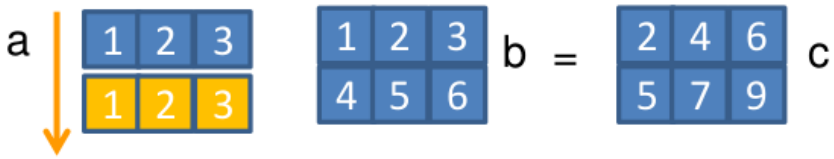


Broadcasting example

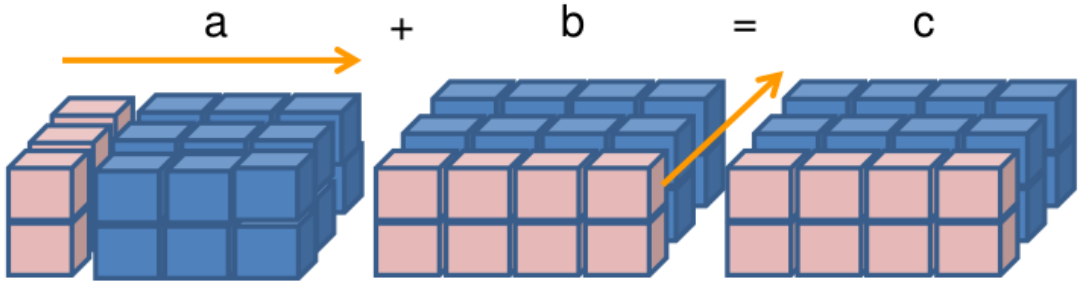
Broadcast can always be used on 1-dimensional arrays.

Examples:

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




```
a=np.arange(30)
a=a.reshape((2,5,3))
b=np.arange(8)
b=b.reshape((2,4,1))
c=a+b      # No Broadcastable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,5,3) (2,4,1)
```

Vectorization

For loops are slow in Python.
One advatage in using numpy arrays is the provided ability to execute a lot of operations avoiding explicit loops.
Avoiding explicit loops is called vectorization.

Example:
Esempi:
```
a=np.arange(0,4*np.pi,0.1)
```

VECTORIZED VERSION
```
y=np.sin(a)*2
```

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

Sometimes it is needed to vectorize explicitely the algorithm:
- Directly: vectorize(function) # a bit slow!
- Manually: withsuitable techniques, like slicing for example

Vectorization

Vectorization is not always possible.

Example:

```
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()
```

- Scalar version to work with arrays:

```
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
```

This implementation is very slow in Python and it works only for 1-dimensional arrays

=> The 'where' statement can be used instead

Vectorization

```
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)
```

Using vectorization, the previous examples becpomes:

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

- Avoid for cicle usage
- Run on molti-dimentional structures

This is the famous pythonic way of work

Vectorization

Array slicing can be used to vectorize operations.

In scientific field, for example, applications regarding
- schemas for finite differences equations
- image processing

it is common to find expressions like:

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

It can be managed:

- with scalar functions

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

- or using vectorization:

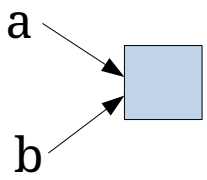
```
x[1:n-1]=x[0:n-2]+2*x[1:n-1]+x[2:n]
```

Array copy

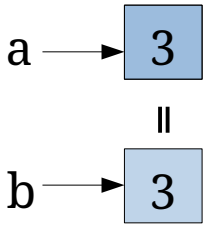
Copy can be of two types:

- copy by reference (it is the copy of memory area pointer)

a = b means:

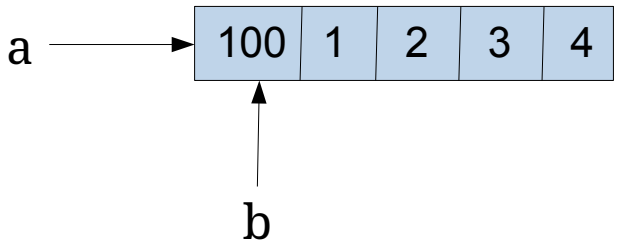


- copy by value (a new memori area is crested with the same value)



Array copy is by default by reference:

```
>>>a=arange(5)
a: [0,1,2,3,4]
>>>b=a
>>>b[0]=100
>>>print “a:”, a , “b:” , b
a: [100,1,2,3,4]          b: [100,1,2,3,4]
```



Array assignment by value is done using method copy:

```
>>>c=a.copy()
>>>print “id(a): “, id(a), “id(c):”, id(c)
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]
```

I/O with array NumPy

Functions eval and repr can be used to write and read ASCII format files

```
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()
```

Files I/O can be managed with loadtxt and savetxt

Write file:

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

Read file:

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

I/O with array NumPy

Text.txt

Student	test1	test2		test3	test4
Lisa	98.3	94.2	95.3	91.3	
Carlo	47.2	49.1	54.2	34.7	
Mario	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]]
```

Matrix

Numpy provides standard classes, inheriting by array and using its internal structure

- Matrix inherit from ndarray methods and attributes
- Matrix class specific attributes
  - .T trasposta
  - .H coniugata trasposta
  - .I inversa
  - .A array bidimensionale
- Matrix defines only bidimensional objects
- Matrix * operator executes multiplication
- Matrix objects have priority respect to simple arrays

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

The Numpy module contains interesting submodules. One of them is

linalg

containing some algorithm of linear algebra.
It contains functions to solve:

- linear systems
- compute eigenvalues
- compute eigenvectors
- factorization
- invert matrix
- matrix multiply

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

linalg: example

```
>>> 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 is another NumPy sub-module to generate random numbers

>>> dir(random)

The standard numpy module is not efficient in random number ganeration, it is more efficient to use numpy.random

Example:

```
>>> 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)
```