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

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(x.size):
        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 np.where(condition, x1, x2)
```

Using vectorization, the previous examples becomes:

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

- Avoid for cicle usage
- Run on multi-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]
```

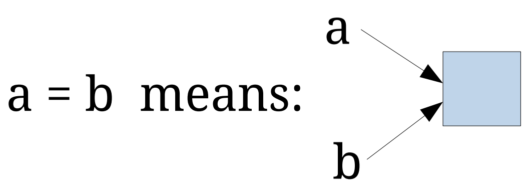
- or using vectorization:

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

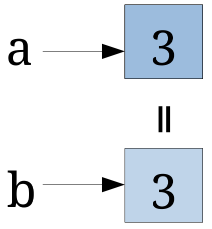
Array copy

Copy can be of two types:

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

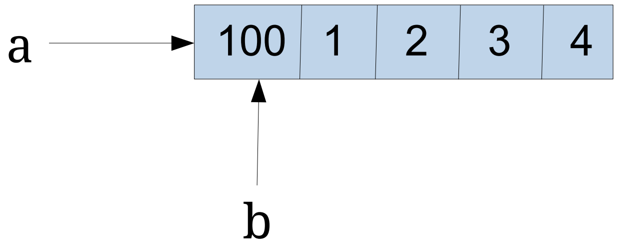


- copy by value (a new memory area is created with the same value)



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

Read file:

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

Write file:

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

I/O with array NumPy

```
$ cat textfile.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 = np.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 = np.loadtxt(‘textfile.txt’,skiprows=2,usecols=(1,-2))
>>> print (b)
[[ 98.3  95.3]
 [ 47.2  54.2]
 [ 84.2  94.1]]
```

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)
```

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, IDL.

https://docs.scipy.org/doc/scipy/reference/tutorial/index.html

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

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. For the power user, you have 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.

Matplotlib: Gallery

https://matplotlib.org/gallery/index.html#examples-index

This gallery contains examples of the many things you can do with Matplotlib.

It is completely searchable from the search page:

https://matplotlib.org/search.html

A set of tutorial is accessible:

https://matplotlib.org/tutorials/index.html

example code: simple_plot.py

Simple plot of a sin function, with labels on x and y axis (simple_plot.py):

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

t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2*np.pi*t)
plt.plot(t, s)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('About as simple as it gets, folks')
plt.grid(True)
plt.savefig("test.png")
plt.show()
```

https://matplotlib.org/examples/pylab_examples/simple_plot.html

Exercise

Using the previous example, make some try changing the scale and the labels.

Try to plot also different functions.

Example: subplots

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

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'o-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.show()
```

https://matplotlib.org/gallery/subplots_axes_and_figures/subplot.html

Example: statistics

```
import numpy as np
from matplotlib import pyplot as plt

# read data by file
data = np.loadtxt('data/populations.txt')

# read variables by line
year, hares, lynxes, carrots = data.T

# plot populations
print("plot the 4 populations on the same graph")
plt.axes([0.2, 0.1, 0.5, 0.8])
plt.plot(year, hares, year, lynxes, year, carrots)
plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
plt.show()
plt.close()
```

```
print("The mean populations over time:")
populations = data[:, 1:]
print(populations.mean(axis=0))
# Expected result:
# [ 34080.95238095  20166.66666667  42400.    ]

print("The sample standard deviations:")
print(populations.std(axis=0))

# Expected result:
# [ 20897.90645809 16254.59153691 3322.5062]

print("Which species has the highest population
      each year?:")
print(np.argmax(populations, axis=1))

# Expected result:
# [2 2 0 0 1 1 2 2 2 2 2 2 0 0 0 1 2 2 2 2 2]
```

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