Introduction to Scientific Computing with Python

Chiara Ravazzi

National Research Council of Italy
Institute of Electronics, Computer and Telecommunication Engineering, c/o Politecnico di Torino, Italy
Systems Modeling & Control Group

Analisi tempo-frequenza e multiscala – 01RMQNG

Python

About Python :

- high-level programming language
- simple to use and learn with clean and concise syntax
- pseudo-compiled language (no separate compilation phase)
- interpreted and executed on most of the platforms (Mac, Microsoft Windows, Linux)

Why Python :

- open-source software
- huge and dynamic international community (no-profit Python Software Foundation)
- one of main technologies of the core business of big companies (e.g. Google, Industrial Light & Magic)
- General Purpose language (well suited for scientific computing)

`http://www.python.it/`

Tools

Many libraries and packages available

- Numpy : fundamental package for scientific computing with Python
- Scipy : collection of numerical methods and tools for advanced scientific computing
- Matplotlib : high quality plotting library

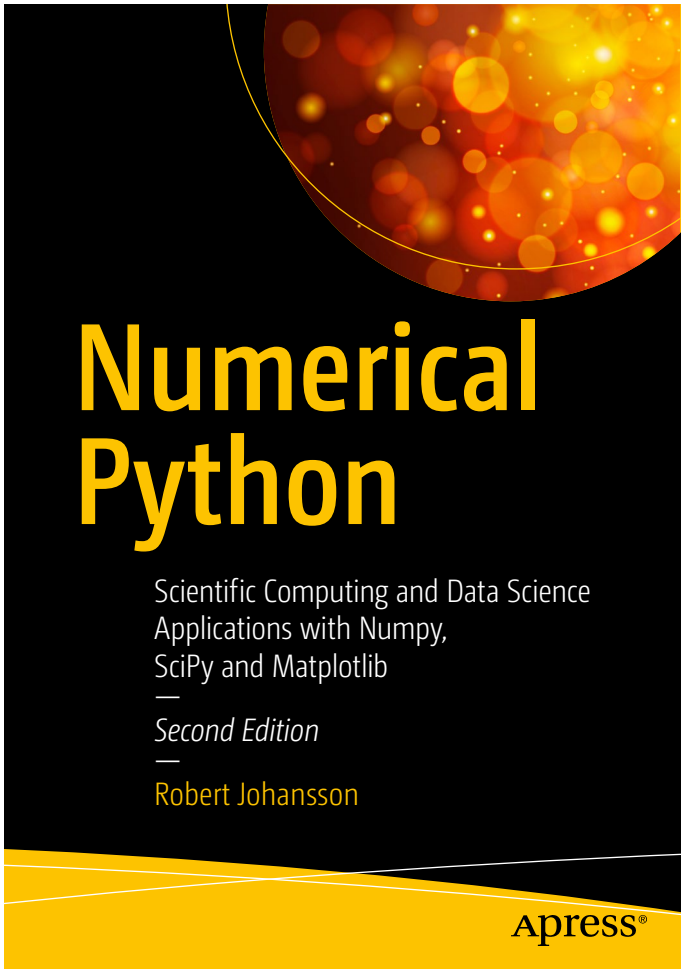
Anaconda distribution of Python

`https://www.anaconda.com/distribution`

- includes different packages
- has a graphical interface (IDE Spyder)
- well suited for scientific calculation and data analysis.
- download (free of charge) from the site :

`https://www.continuum.io`

Numerical Python



A code sample

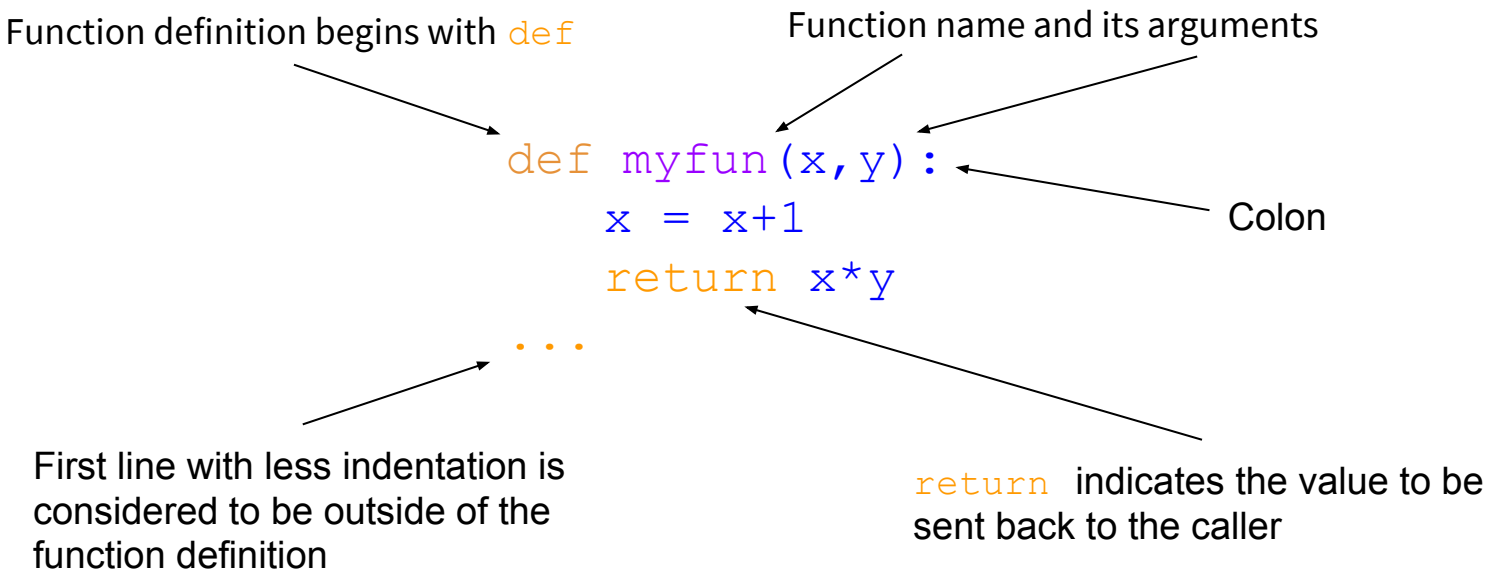
```
x = 34 - 23    # A comment.
y = "Hello"    # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + "World" # String concat.
print(x)
print(y)
```

Python – Preliminaries

Enough to understand the code

- Indentation matters to the meaning of the code
  - No braces {} to mark blocks of code in Python, use consistent indentation instead
  - The first line with less indentation is outside of the the block
  - The first line with more indentation starts a nested block
- The first assignment to a variable creates it
  - Variable types do not need to be declared
  - Python figures out the variable types on its own
- Assignment uses = and comparison uses ==
- For numbers + - * / **
- Logical operators are words (and, or, not) not symbols
- The basic printing command is print()

Functions : syntax



How to write a function

- any text editor can be used
- Spyder has its own text editor (colors, automatic indentation, semantic check)
- save myfun.py file
- a file .py is a module (it can contain more than one function)

How to load a function

- load into memory (more precisely into the namespace of the main module).
- example : the module programs contains the functions myfun1 and myfun2
  - import programs
    ... programs.myfun1() ... programs.myfun2() ...
  - import programs as pr
    ... pr.myfun1() ... programs.myfun2() ...
  - from programs import myfun1, myfun2
    ... myfun1() ... myfun2() ...
  - from programs import *
    ... myfun1() ... myfun2() ...

For Loops

A for loop steps through each of the items in a collection type, or any type of object which is iterable

```
for <item> in <collection>:
    <statements>
```

we often want to write a loop where the variable ranges over some sequence of numbers. The `range()` function returns a list of numbers from 0 up to but *not including* the number we pass to it.

```
for x in range(5):
    print(x)
```

Similarly **if** and **while**

```
def segno(x):
    if x>0:
        s=1
    elif x==0:
        s=0
    else:
        s=-1
    return s
```

```
>>> somma=0; i=0
>>> while somma<=354:
        i=i+1
        somma=somma+i
>>> print(i)
27
```

Numpy

About Numpy

- Library providing tools for the management of arrays, matrices, and multidimensional array
- Download from : `http://numpy.scipy.org/`
- Importing the library : `import numpy as np`
- Provide high-level functions (e.g. linear algebra) :
  `from numpy import linalg as la`

Numpy – Array

About numpy arrays

- homogeneous, i.e. of the same type (dtype)
- typed arrays of fixed size
- creation directly from a list or from a tuple

Basic attributes of the *ndarray* class

Attribute	Description
`Shape`	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
`Size`	The total number elements in the array.
`Ndim`	Number of dimensions (axes).
`nbytes`	Number of bytes used to store the data.
`dtype`	The data type of the elements in the array.

Numpy – Array

```
In [2]: data = np.array([[1, 2], [3, 4], [5, 6]])
In [3]: type(data)
Out[3]: <class 'numpy.ndarray'>
In [4]: data
Out[4]: array([[1, 2],
               [3, 4],
               [5, 6]])
In [5]: data.ndim
Out[5]: 2
In [6]: data.shape
Out[6]: (3, 2)
In [7]: data.size
Out[7]: 6
```

Numpy – Array

Data types

dtype	Variants	Description
`int`	`int8, int16, int32, int64`	Integers
`uint`	`uint8, uint16, uint32, uint64`	Unsigned (nonnegative) integers
`bool`	`Bool`	Boolean (True or False)
`float`	`float16, float32, float64, float128`	Floating-point numbers
`complex`	`complex64, complex128, complex256`	Complex-valued floating-point numbers

```
In [22]: d1 = np.array([1, 2, 3], dtype=float)
In [23]: d2 = np.array([1, 2, 3], dtype=complex)
In [24]: d1 + d2
Out[24]: array([ 2.+0.j,  4.+0.j,  6.+0.j])
In [25]: (d1 + d2).dtype
Out[25]: dtype('complex128')
```

Numpy – Array

Real and Imaginary parts

```
In [28]: data = np.array([1, 2, 3], dtype=complex)
In [29]: data
Out[29]: array([ 1.+0.j,  2.+0.j,  3.+0.j])
In [30]: data.real
Out[30]: array([ 1.,  2.,  3.])
In [31]: data.imag
Out[31]: array([ 0.,  0.,  0.])
```

Numpy – Array

Numpy functions for generating arrays

Function Name	Type of Array
`np.array`	Creates an array for which the elements are given by an array-like object, which, for example, can be a (nested) Python list, a tuple, an iterable sequence, or another `ndarray` instance.
`np.zeros`	Creates an array with the specified dimensions and data type that is filled with zeros.
`np.ones`	Creates an array with the specified dimensions and data type that is filled with ones.
`np.diag`	Creates a diagonal array with specified values along the diagonal and zeros elsewhere.
`np.arange`	Creates an array with evenly spaced values between the specified start, end, and increment values.
`np.linspace`	Creates an array with evenly spaced values between specified start and end values, using a specified number of elements.
`np.logspace`	Creates an array with values that are logarithmically spaced between the given start and end values.

Numpy – Array

```
In [35]: np.array([[1, 2], [3, 4]])
Out[35]: array([[1,  2],
                [3, 4]])
In [36]: data.ndim
Out[36]: 2
In [37]: data.shape
Out[37]: (2, 2)

In [38]: np.zeros((2, 3))
Out[38]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.]])
In [39]: np.ones(4)
Out[39]: array([ 1.,  1.,  1., 1.])
```

Numpy – Array

Arrays filled with Incremental Sequences

```
In [51]: np.arange(0.0, 10, 1)
Out[51]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
In [52]: np.linspace(0, 10, 11)
Out[52]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.])
```

Creating matrix arrays

```
In [62]: np.identity(4)
Out[62]: array([[ 1.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.]])

In [65]: np.diag(np.arange(0, 20, 5))
Out[65]: array([[0,  0,  0,  0],
                [0,  5,  0,  0],
                [0,  0, 10,  0],
                [0,  0,  0, 15]])
```

Numpy – Array

Come estrarre gli elmeenti da un array

Indexing and slicing

- Elements access using square brackets
- Index starts at 0
- Slices specified by :

Indexing and Slicing expressions

Expression	Description
`a[m]`	Select element at index *m*, where *m* is an integer (start counting form 0).
`a[-m]`	Select the *n*th element from the end of the list, where *n* is an integer. The last element in the list is addressed as –1, the second to last element as –2, and so on.
`a[m:n]`	Select elements with index starting at *m* and ending at *n* – 1 (*m* and *n* are integers).
`a[:]` or `a[0:-1]`	Select all elements in the given axis.
`a[:n]`	Select elements starting with index 0 and going up to index *n* – 1 (integer).
`a[m:]` or `a[m:-1]`	Select elements starting with index *m* (integer) and going up to the last element in the array.
`a[m:n:p]`	Select elements with index *m* through *n* (exclusive), with increment *p*.
`a[::-1]`	Select all the elements, in reverse order.

Numpy – Array

Indexing and slicing

```
In [66]: a = np.arange(0, 11)
In [67]: a
Out[67]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

In [68]: a[0]  # the first element
Out[68]: 0
In [69]: a[-1] # the last element
Out[69]: 10
In [70]: a[4]  # the fifth element, at index 4
Out[70]: 4

In [71]: a[1:-1]
Out[71]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
In [72]: a[1:-1:2]
Out[72]: array([1, 3, 5, 7, 9])
```

Numpy – Array

Indexing and slicing

```
In [73]: a[:5]
Out[73]: array([0, 1, 2, 3, 4])
In [74]: a[-5:]
Out[74]: array([6, 7, 8, 9, 10])

In [75]: a[::-2]
Out[75]: array([10,  8,  6,  4,  2,  0])
```

Numpy – Multidimensional Arrays

Views

```
In [78]: A
Out[78]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])
In [85]: B = A[1:5, 1:5]
In [86]: B
Out[86]: array([[11, 12, 13, 14],
                [21, 22, 23, 24],
                [31, 32, 33, 34],
                [41, 42, 43, 44]])
In [87]: B[:, :] = 0
In [88]: A
Out[88]: array([[ 0,  1,  2,  3,  4,  5],
                [10,  0,  0,  0,  0, 15],
                [20,  0,  0,  0,  0, 25],
                [30,  0,  0,  0,  0, 35],
                [40,  0,  0,  0,  0, 45],
                [50, 51, 52, 53, 54, 55]])
```

Copies

```
In [89]: C = B[1:3, 1:3].copy()
In [90]: C
Out[90]: array([[0, 0],
                [0, 0]])
In [91]: C[:, :] = 1  # this does not affect B since C is a copy of the
view B[1:3, 1:3]
In [92]: C
Out[92]: array([[1, 1],
                [1, 1]])
In [93]: B
Out[93]: array([[0, 0, 0, 0],
                [0, 0, 0, 0],
                [0, 0, 0, 0],
                [0, 0, 0, 0]])
```

Visual summary of indexing methods

data

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[0]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[1, :]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[:, 2]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[0:2, 0:2]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[0:2, 2:4]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[::2, ::2]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[1::2, 1::2]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[:,[0,3]]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[[1,3],[0,3]]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[:,np.array([False, True, True, False])]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

data[1:3,np.array([False, True, True, False])]

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

Numpy – Multidimensional Arrays

Reshaping and resizing

Function/Method	Description
`np.reshape`, `np.ndarray.reshape`	Reshape an N-dimensional array. The total number of elements must remain the same.
`np.ndarray.flatten`	Creates a copy of an N-dimensional array, and reinterpret it as a one-dimensional array (i.e., all dimensions are collapsed into one).
`np.ravel`, `np.ndarray.ravel`	Create a view (if possible, otherwise a copy) of an N-dimensional array in which it is interpreted as a one-dimensional array.
`np.squeeze`	Removes axes with length 1.
`np.expand_dims`, `np.newaxis`	Add a new axis (dimension) of length 1 to an array, where `np.newaxis` is used with array indexing.
`np.transpose`, `np.ndarray.transpose`, `np.ndarray.T`	Transpose the array. The transpose operation corresponds to reversing (or more generally, permuting) the axes of the array.

Numpy – Multidimensional Arrays

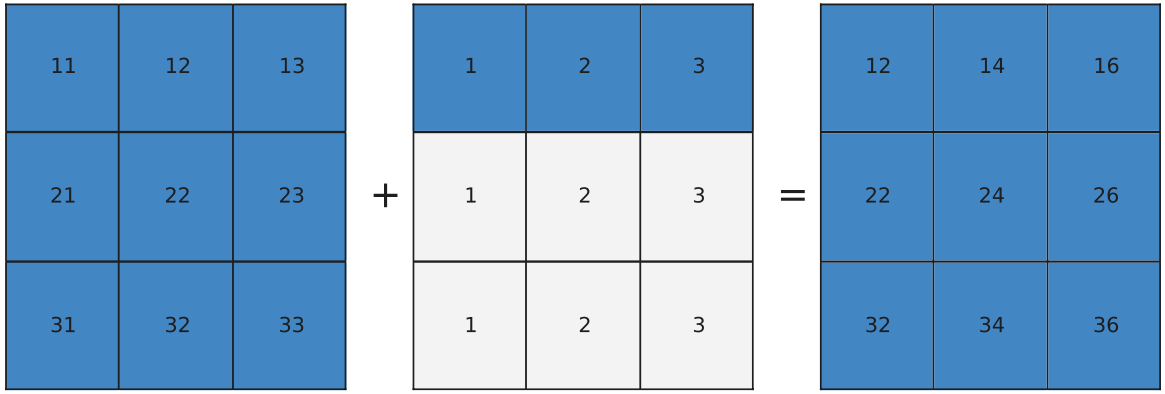
```
In [112]: data = np.array([[1, 2], [3, 4]])
In [113]: np.reshape(data, (1, 4))
Out[113]: array([[1, 2, 3, 4]])
In [114]: data.reshape(4)
Out[114]: array([1, 2, 3, 4])
```

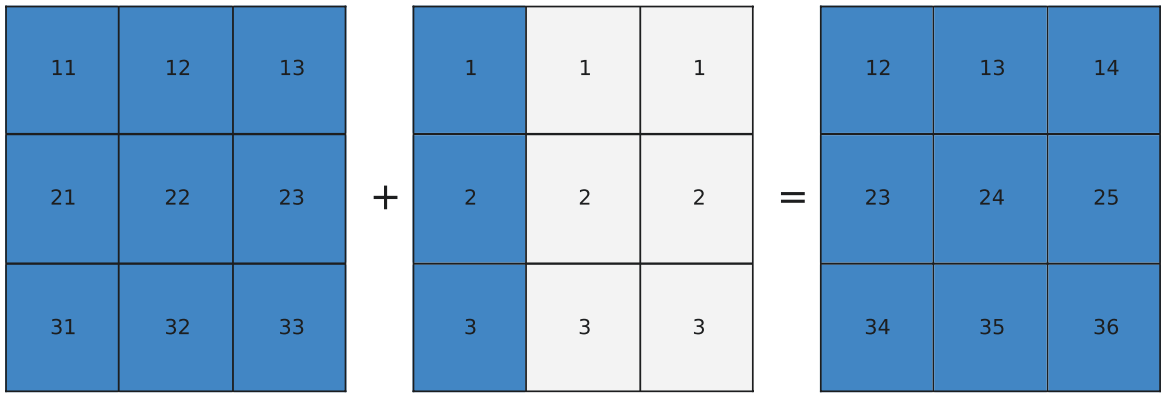
Elementwise arithmetic operators

Operator	Operation
+, +=	Addition
-, -=	Subtraction
*, *=	Multiplication
/, /=	Division
//, //=	Integer division
**, **=	Exponentiation

Numpy – Elementwise Functions

Broadcasting





Numpy – Elementwise Functions

Elementwise elementary mathematical functions

NumPy Function	Description
`np.cos, np.sin, np.tan`	Trigonometric functions.
`np.arccos, np.arcsin, np.arctan`	Inverse trigonometric functions.
`np.cosh, np.sinh, np.tanh`	Hyperbolic trigonometric functions.
`np.arccosh, np.arcsinh, np.arctanh`	Inverse hyperbolic trigonometric functions.
`np.sqrt`	Square root.
`np.exp`	Exponential.
`np.log, np.log2, np.log10`	Logarithms of base e, 2, and 10, respectively.

Numpy – Elementwise Functions

```
In [159]: x = np.linspace(-1, 1, 11)
In [160]: x
Out[160]: array([-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1.])
In [161]: y = np.sin(np.pi * x)
In [162]: np.round(y, decimals=4)
Out[162]: array([-0., -0.5878, -0.9511, -0.9511, -0.5878, 0., 0.5878, 0.9511,
          0.9511, 0.5878, 0.])
```

Numpy – Elementwise functions

Elementwise Mathematical Operations

NumPy Function	Description
`np.add, np.subtract, np.multiply, np.divide`	Addition, subtraction, multiplication, and division of two NumPy arrays.
`np.power`	Raises first input argument to the power of the second input argument (applied elementwise).
`np.remainder`	The remainder of division.
`np.reciprocal`	The reciprocal (inverse) of each element.
`np.real, np.imag, np.conj`	The real part, imaginary part, and the complex conjugate of the elements in the input arrays.
`np.sign, np.abs`	The sign and the absolute value.
`np.floor, np.ceil, np.rint`	Convert to integer values.
`np.round`	Rounds to a given number of decimals.

Numpy – Elementwise functions

Aggregate functions

NumPy Function	Description
`np.mean`	The average of all values in the array.
`np.std`	Standard deviation.
`np.var`	Variance.
`np.sum`	Sum of all elements.
`np.prod`	Product of all elements.
`np.cumsum`	Cumulative sum of all elements.
`np.cumprod`	Cumulative product of all elements.
`np.min, np.max`	The minimum/maximum value in an array.
`np.argmin, np.argmax`	The index of the minimum/maximum value in an array.
`np.all`	Returns True if all elements in the argument array are nonzero.
`np.any`	Returns True if any of the elements in the argument array is nonzero.

Numpy – Elementwise functions

Functions for conditional and logical expressions

Function	Description
`np.where`	Chooses values from two arrays depending on the value of a condition array.
`np.choose`	Chooses values from a list of arrays depending on the values of a given index array.
`np.select`	Chooses values from a list of arrays depending on a list of conditions.
`np.nonzero`	Returns an array with indices of nonzero elements.
`np.logical_and`	Performs an elementwise AND operation.
`np.logical_or, np.logical_xor`	Elementwise OR/XOR operations.
`np.logical_not`	Elementwise NOT operation (inverting).

Numpy – Functions

Functions for matrix operations

NumPy Function	Description
`np.dot`	Matrix multiplication (dot product) between two given arrays representing vectors, arrays, or tensors.
`np.inner`	Scalar multiplication (inner product) between two arrays representing vectors.
`np.cross`	The cross product between two arrays that represent vectors.
`np.tensordot`	Dot product along specified axes of multidimensional arrays.
`np.outer`	Outer product (tensor product of vectors) between two arrays representing vectors.
`np.kron`	Kronecker product (tensor product of matrices) between arrays representing matrices and higher-dimensional arrays.
`np.einsum`	Evaluates Einstein's summation convention for multidimensional arrays.

Matplotlib

About Matplotlib

- Extension providing tools for creating 2D and 3D graphs
- Numpy module for the management of high-dimensional data
- Download from : `http://www.matplotlib.org`
- Plotting functions in pyplot module
- Importing the library : `import matplotlib.pylab as plt`

Basics

- Plot a function :

```
plt.plot(x, y, 'r-')
plt.show()
```

- Display more than one curve

```
plt.plot(x1, y1, 'r-',x2, y2, 'b-')
plt.show()
```

or

```
plt.plot(x1, y1, 'r-')
plt.plot(x2, y2, 'b-')
plt.show()
```

- Additional paramters : linestyle, linewidth, marker, markersize, label
- Display legend : `pl.legend()`

Scypy

About Scypy : Library including modules for

- optimization
- linear algebra
- integration
- functions for signal and image processing
- Downloadable from : `http://www.scipy.org`
- Importing the library : `from scipy import linalg`

Scypy – Linear algebra

Basic functions : `inv, solve, det, norm, lstsq, pinv`

```
>>> A = numpy.array([[1,2], [3,4]])
>>> linalg.inv(A)
    array([[-2. ,  1. ],
           [ 1.5, -0.5]])
#verifica: A·A^-1 = I
>>> A.dot(linalg.inv(A))
    array([[1.000e+00,0.000e+00],
          [4.441e-16, 1.000e+00]])

>>> A = linalg.det(A)
    -2
```

Scypy – Linear algebra

Decomposition : `eig, lu, svd, orth, cholesky, qr, schu`

```
>>> a = numpy.array([[1,3,5], [2,5,1],
[2,3,6]])

#calcolo autovalori e autovettori
>>> aval, avec = linalg.eig(a)
>>> aval
    array([ 9.39895873+0.j,
           -0.73379338+0.j,
            3.33483465+0.j])
#gli autovettori sono le colonne della
matrice "avec". Es. primo autovettore
>>> avec[:,0]
    array([-0.57028326,
           -0.41979215,
           -0.70608183])
```

Scypy – Linear algebra

Solving linear systems:

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

#soluzioni
>>> x = linalg.solve(A,b)
>>> x
    array([[-4. ],
           [ 4.5]])
#verifica
>>> A.dot(x)-b
    array([[ 0.],
           [ 0.]])
```