Python Lecture 4 – Python libraries usage: numpy, matplotlib e scipy examples

- Numpy
- Scipy
- Matplotlib
- Exceptions
- Classes

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/abilita_info_units_2021

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 to 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.

ndarray efficiency

In NumPy 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 the operation at near-C speeds

Vectorization and broadcasting

Vectorization and broadcasting are two of NumPy’s features which are the basis of much of its power.

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/dimentions of the array

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

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

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

• L’attributo ndim specifica la dimensione dell’array

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

numpy array glossary (2)

itemsize allows to specify the dimension of each array element.

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

array creation

A NumPy array can be created by an object

Example:

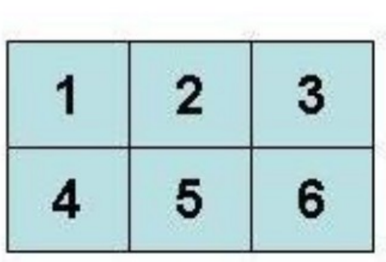
```
>>>import numpy as np
>>>a = np.array([1,2,3,4])
>>>list1 = [1,2,3,4]
>>>tupla = (5,6,7,8)
>>>a = np.array(list)              # from a list
>>>b = np.array(tupla)             # from a tupla
>>>c = np.array([list1,tupla])     # from a list and from a tupla
>>> c
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>>a.dtype                         # check the array type
dtype('int32')
```

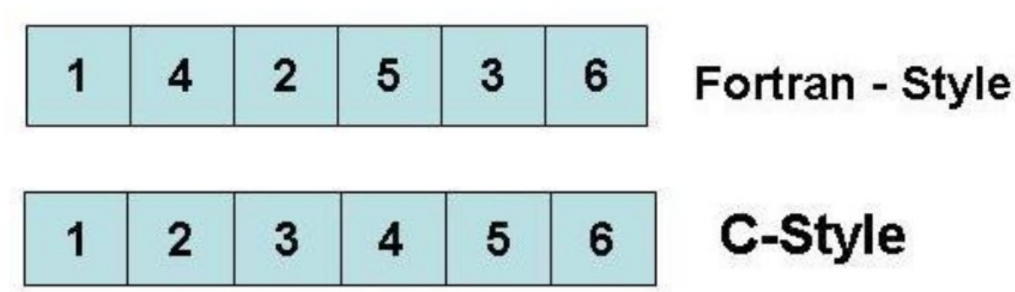
array memory allocation

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:

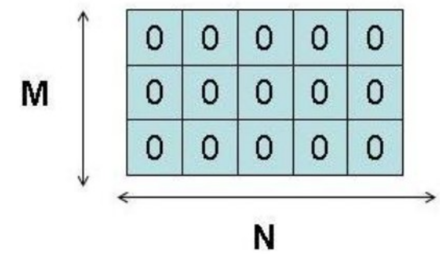




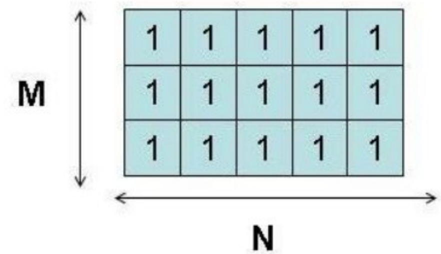
Other array creations

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

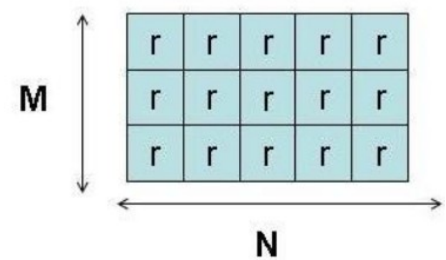
- zeros(shape, dtype=float, order ='C') function
  create an array of 0 of shape dimension



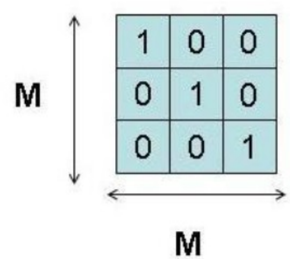
- ones(shape,dtype=None, order ='C')
  create an array of 1 of shape dimension



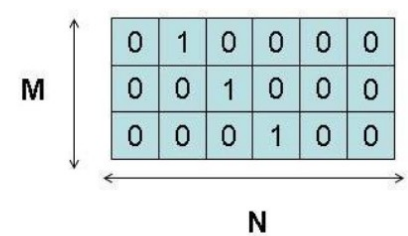
- empty(shape, dtype=None, order ='C')
  creates an array with shape dimension without initializing it



- 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()

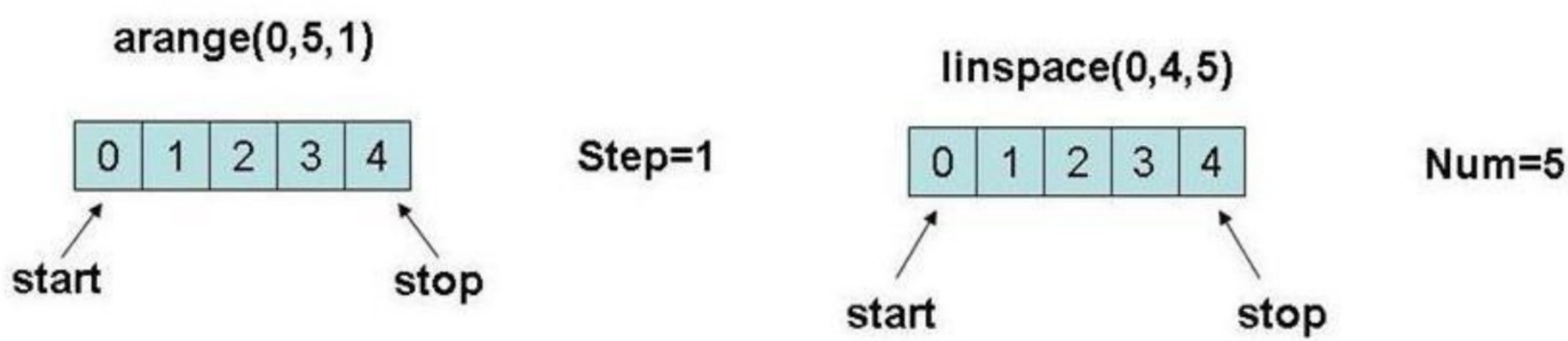
An array can be created from a numbers sequence with functions similar to function range() for lists:

=> 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, restep=False)

creates a sequence of *num* numbers uniformely distributed between *start* and *stop,*
If endpoint=True, stop is the last sample; If restep=True, return (samples, step)



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.)

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

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("Autoincrement b +=1 b=", b)
print("Multiply c*3 " ,c, "* 3= ",c*3)
print("Sin (c)", np.sin(c))
```

Output:

```
Sum [5,6,7,8] + [1,2,3,4] = [6,8,10,12]
Autoincrement 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 on arrays

Product vector-matrices

Given two vectors

```
v1=np.array([1,2,3])
v2=np.array([10,20,30])
```

product element by element between monodimensional array

```
v1*v2
```

<u>Output:</u>

```
array([10, 40, 90])
```

scalar product between monodimensional array

```
np.dot(v1,v2)
```

<u>Output:</u>

```
140
```

Numerical operations on arrays

product between matrices

use the np.matrix type

```
m1=np.matrix(v1)
m2=np.matrix(v2)
```

are bidimensional arrays:

```
m1.shape,m2.shape
```

Output:

((1, 3), (1, 3))

You can use standard operators like in traditional linear algebra:

```
try:
    m1*m2 #ERRORE
except Exception as err:
    print(err)
```

Output:

shapes (1,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)

Re-define m2 as column vector:

```
m2=np.matrix(v2[:,np.newaxis])
```

re-try:

```
m1*m2
```

Output:

matrix([[140]])

The same doing:

```
np.dot(m1,m2)
```

Output:

matrix([[140]])

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 output

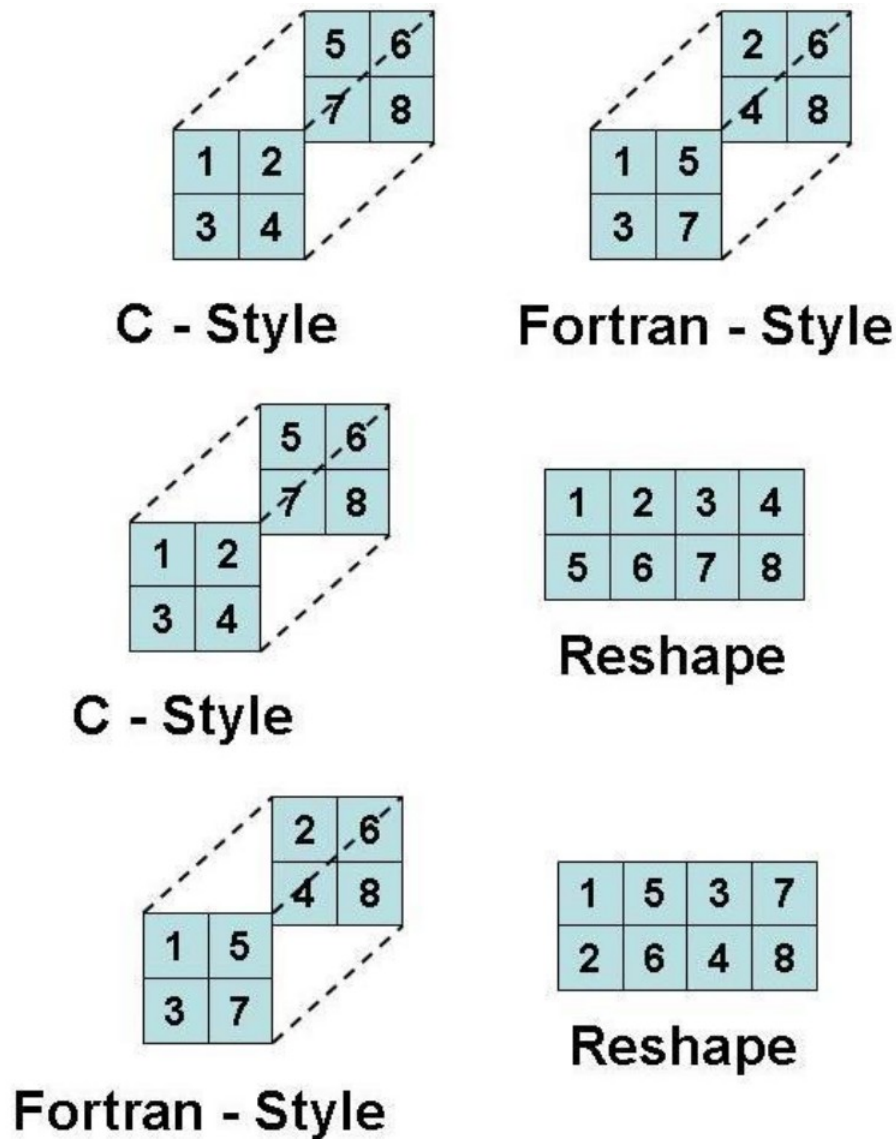
Shape (8,)

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]]



reshape() example (try)

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

>>> print("Shape" , a.shape)

>>> c_style = a.reshape((2,2,2),order='C')          # Array Method: C Style

>>> f_style = a.reshape((2,2,2),order='F')          # Array Method: Fortran Style

>>> 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)
```

indexing – slicing – iteration (1)

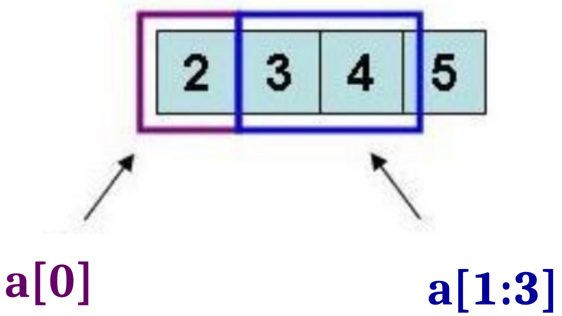
The access to array elements is done by the operator[]

array has the slicing operator[:]

In case of monodimensional arrays the built-in list notation

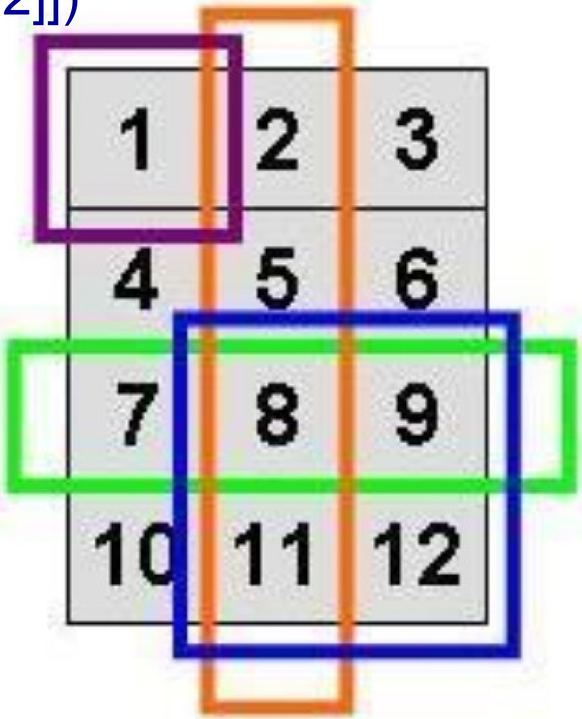
Example

```
>>> 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]])
```

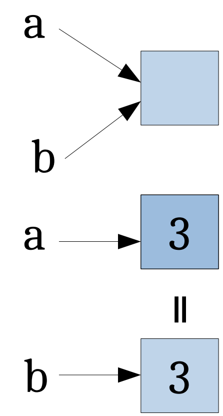
```
>>> print(a[::,1])
array([ 1, 6, 11, 16, 21])
>>> print(a[1])
array([5, 6, 7, 8, 9])
>>> print(a[1,:])
array([5, 6, 7, 8, 9])
>>> print(a[1,::])
array([5, 6, 7, 8, 9])
>>> print(a[1,::2])
array([5, 7, 9])
>>> print(a[1,10::-1])
array([9, 8, 7, 6, 5])
```

Array copy

Copy can be of two types:

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

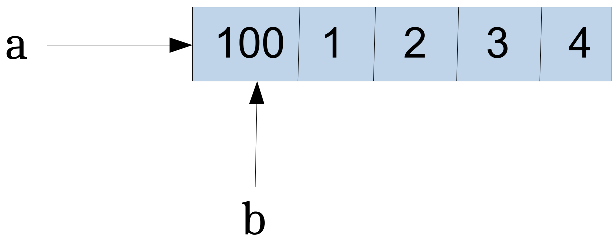
a = b means:



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

<u>Array copy is by default by reference</u>:

```
>>>a=np.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]
```



<u>Array assignment by value</u> 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]=122
>>> print("c" , c , "a", a)
c [122, 1, 2, 3, 4] a [100, 1, 2, 3, 4]
```

Copy element by element is by value

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
>>> b[3] = 1000
>>> b == a
array([True,True,True,False,True],dtype=bool)
```

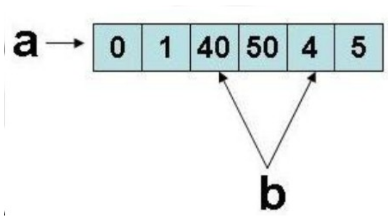
Slicing is by reference

Note:

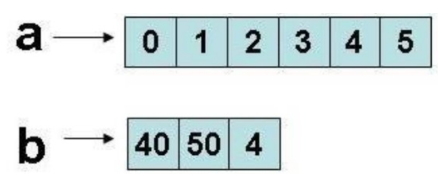
The slicing operation for numpy arrays is different from slicing for python built-in lists:

- in numpy array slicing the generated sub-array is a reference to the original memory area
- 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]
```



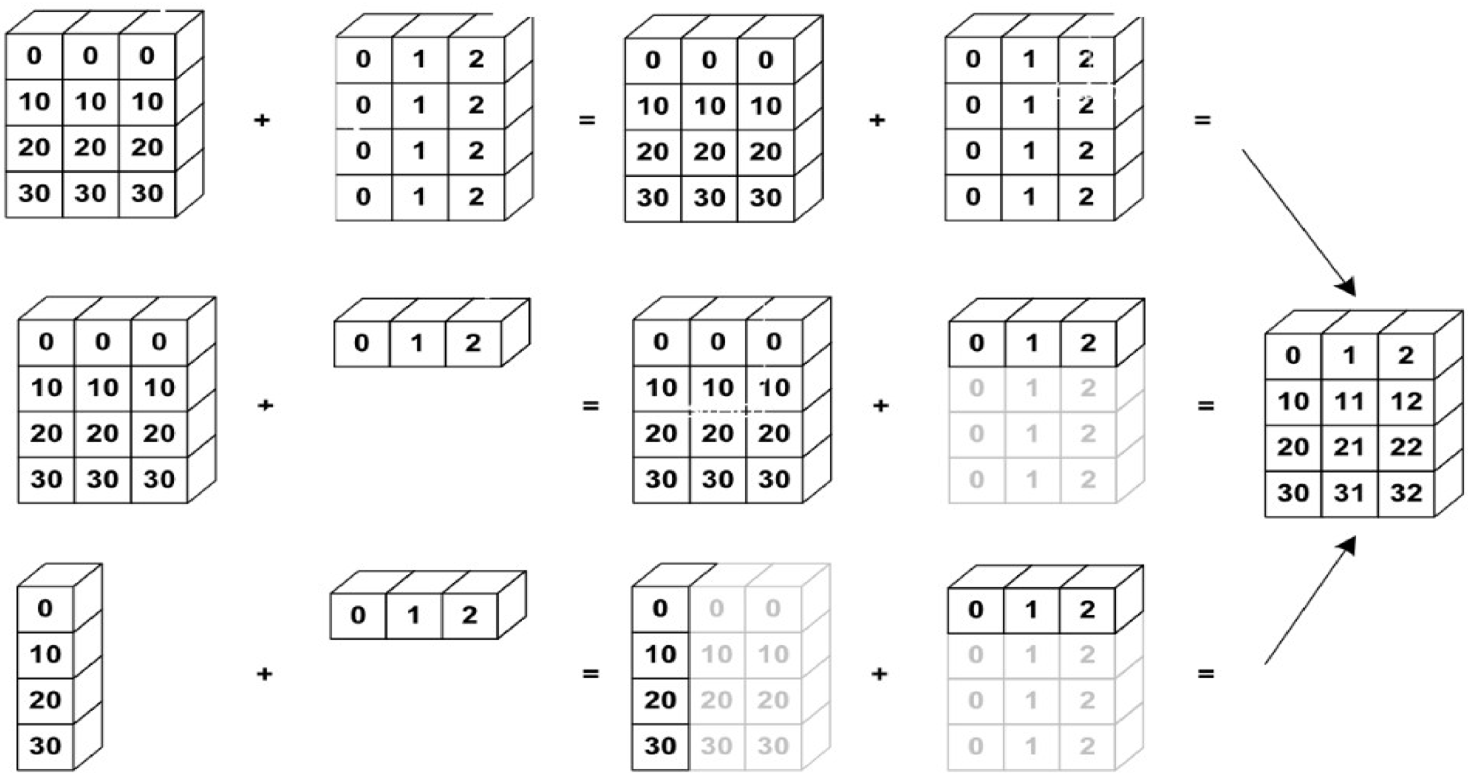
This impacts on performances and memory consumption and results.

Broadcasting

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

This works on arrays of the same size.

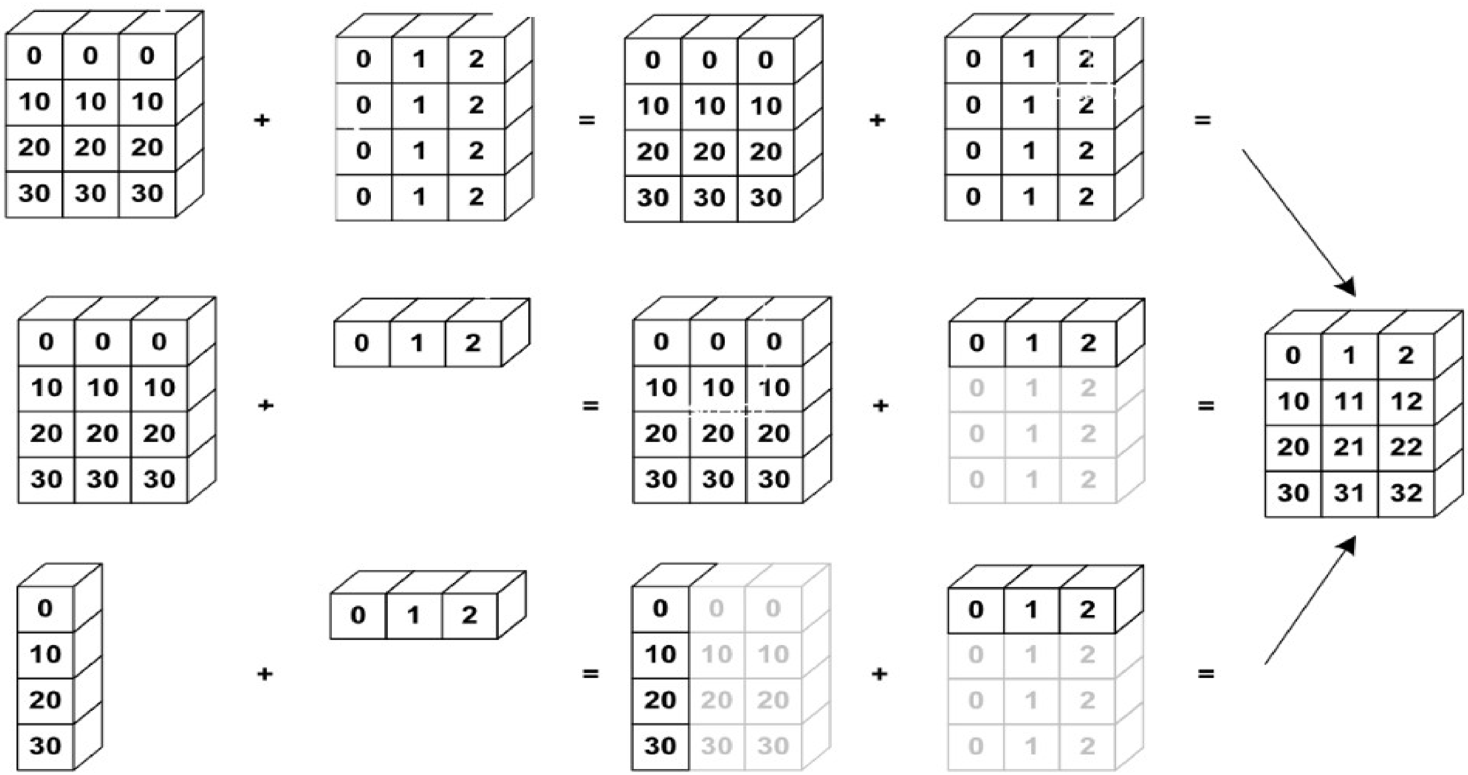
Nevertheless, It’s also possible to do operations on arrays of different sizes if NumPy can transform these arrays so that they all have the same size: this conversion is called broadcasting.
The image below gives an example of broadcasting:



Broadcasting rules

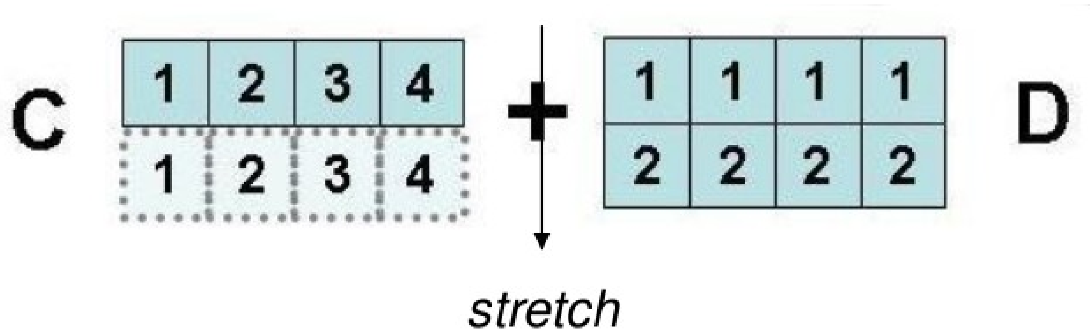
The broadcasting has two rules:

- 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

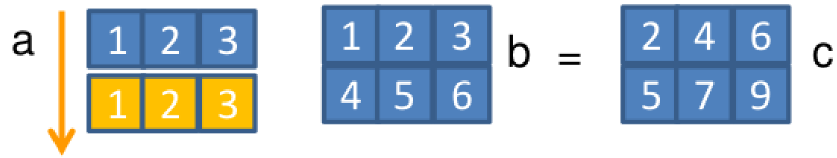
```
c=np.arange(1,5)
d=np.array([[1,1,1,1],[2,2,2,2]])
print d, ”+”, c “= “ d+c
```



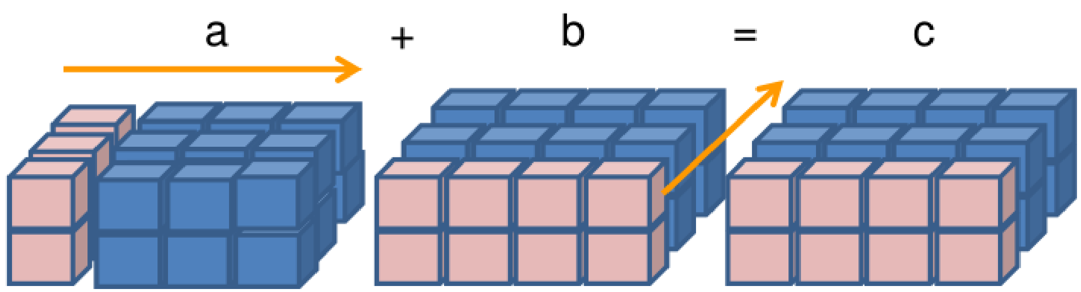
Broadcasting example

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:

```
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 range(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: with suitable techniques, like slicing for example

Vectorization

Vectorization is not always possible. **Example**:

```
def func(x):
    if x<0: return 1
    else: return np.sin(x)
func(3)
func(np.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. **Example**:

```
def func_NumPy(x):
    r = x.copy() # allocate result array
    for i in range(np.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 becomes:

```
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 range(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]
```

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

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)
```

Autovettore e autovalore

Datala matrice A, quadrata di ordine n, esistono

•uno scalareλ

•un vettore (a n componenti) v, non nullo,

tali che, scrivendo v come colonna, risulti

$Av=\lambda v$?

Se si,

λ viene detto **autovalore** di A e

v viene detto **autovettore** di A relativo a λ

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: statisctics

```
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()
```

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

```
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]
```

astropy

Astropy

The astropy package contains key functionality and common tools needed for performing astronomy and astrophysics with Python.

It is at the core of the Astropy Project, which aims to enable the community to develop a robust ecosystem of Affiliated Packages covering a broad range of needs for astronomical research, data processing, and data analysis.

http://docs.astropy.org/en/stable/

Astropy: content

Data structures and transformations

Constants (astropy.constants)
Units and Quantities (astropy.units)
N-dimensional datasets (astropy.nddata)
Data Tables (astropy.table)
Time and Dates (astropy.time)
Astronomical Coordinate Systems (astropy.coordinates)
World Coordinate System (astropy.wcs)
Models and Fitting (astropy.modeling)
Uncertainties and Distributions (astropy.uncertainty)

Files, I/O, and Communication

Unified file read/write interface
FITS File handling (astropy.io.fits)
ASCII Tables (astropy.io.ascii)
VOTable XML handling (astropy.io.votable)
Miscellaneous: HDF5, YAML, ASDF, pickle (astropy.io.misc)
SAMP (Simple Application Messaging Protocol (astropy.samp)

http://docs.astropy.org/en/stable/

Astropy: content

Computations and utilities

Cosmological Calculations (astropy.cosmology)
Convolution and filtering (astropy.convolution)
Data Visualization (astropy.visualization)
Astrostatistics Tools (astropy.stats)

Nuts and bolts

Configuration system (astropy.config)
I/O Registry (astropy.io.registry)
Logging system
Python warnings system
Astropy Core Package Utilities (astropy.utils)
Astropy Testing Tools
Try the development version

http://docs.astropy.org/en/stable/

Exceptions

Errors detected during execution are called exceptions.
Exceptions are errors raised executing a statement or an expression, also in case they are syntactically correct.
Exceptions are not unconditionally fatal: they can be handled in Python programs. Most exceptions are not handled by programs, however, and result in error messages.

Example:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- Exceptions come in different types, and the type is printed as part of the message. Example are ZeroDivisionError, NameError and TypeError.

Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Example:

- Create class

```
class MyClass:
  def __init__(self, name, age):
    self.attribute1 = value1
    self.attribute2 = value2

  def myfunc(self):
    print("Hello my attrib1 is " + self.attribute1)
```

Example:

- Create and use object
-

```
p1 = MyClass()

p1.myfunc()
print(p1.attribute1)
```

Classes: the __init__ object

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated, i.e.every time the class is being used to create a new object.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created.

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

Classes: methods

Classes can also contain methods. Methods in objects are functions that belongs to the object.

Let us create a method in the Person class that prints a greeting, and execute it on the p1 object:

Example

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

User Defined Exceptions

Programs may name their own exceptions by creating a new exception class.

Exceptions handling

Programs can handle exceptions with the following structure

```
try:
    statement(s)
except ExceptionType1:
    statement(s)
except exceptionType2, exceptionType3:
    statement(s)
……
except:
    statement(s)
else:
    statement(s)
finally:
    statement(s)
```

except EceptionType1 is executed if an Exception of Type1 is raised in the try block

except is executed if a not previously catchd exception is thrown

else is executed if no one exception is thrown in try block

finally is always executed

Exceptions handling: try….except clause

The **try** statement works as follows: the try clause (the statement(s) between the try and except keywords) is executed.

If no exception occurs, the except clause is skipped and the execution of the try statement is finished.

If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the **except** clause is executed, and then execution continues after the try statement.

If an exception occurs which does not match the exception named in the except clause, it is passed on to other except statements and at the end, to the generic except clause, if it is present. If no handler is found, it is an unhandled exception and execution stops with a message.

When a try statement has more than one except clause, to specify handlers for different exceptions, at most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.

Exceptions handling: except clause

An **except** clause may name multiple exceptions as a parenthesized tuple. Example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Exceptions handling: else clause

The try … except statement has an optional **else** clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. **Example**:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try … except statement.

Exceptions handling: final clause

The try statement has the **final** optional clause.

The final clause, which is intended to define clean-up actions, is always executed before leaving the try statement, whether an exception has occurred or not.

When an exception has occurred in the try clause and has not been handled by an except clause (or it has occurred in an except or else clause), it is re-raised after the finally clause has been executed.

The finally clause is also executed "on the way out" when any other clause of the try statement is left via a break, continue or return statement.

Exceptions handling: a complete example

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Exceptions handling: the raise statement

The **raise** statement allows the programmer to force a specified exception to occur. For example:

```
>>>
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```