Python in a Nutshell

Part II: NumPy and Matplotlib

Manel Velasco,[1] PhD and Alexandre Perera,[1,2] PhD

[1]Departament d'Enginyeria de Sistemes, Automatica i Informatica Industrial (ESAII)
Universitat Politecnica de Catalunya

[2]Centro de Investigacion Biomedica en Red en Bioingenieria, Biomateriales y Nanomedicina (CIBER-BBN)
Alexandre.Perera@upc.edu Manel.Velasco@upc.edu

Introduction to Python for Engineering and Statistics
Febraury, 2013

Contents I

Outline

1. Introduction to NumPy
   - Overview
   - Arrays
   - Operating with Arrays
   - Advanced Arrays (ndarrays)
   - Advanced Operations
2. Matplotlib
   - Introduction
   - Figures and Subplots
   - Axes and further control of figures

NumPy

Python has built-in:

containers: lists (costless insertion and append), dictionaries (fast lookup)

high-level number objects: integers, floating point

Numpy is:

- extension package to Python for multi-dimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)

Snippet

```
import numpy as np
a = np.array([0, 1, 2, 3])
a
```

NumPy

For example:

An array containing:

- values of an experiment/simulation at discrete time steps
- signal recorded by a measurement device, e.g. sound wave
- pixels of an image, grey-level or colour
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan
- ...

Memory-efficient container that provides fast numerical operations.

```
In [1]: l = range(1000)
In [2]: %timeit [i**2 for i in l]
1000 loops, best of 3: 403 us per loop
In [3]: a = np.arange(1000)
In [4]: %timeit a**2
100000 loops, best of 3: 12.7 us per loop
```

NumPy

In case you need help...

- Interactive help

```
help(np.array)
Help on built-in function array in module numpy.core.multiarray:
array(...)
    array(object, dtype=None, copy=True, order=None, subok=False, ...
...
```

```
In [5]: np.array ?
String Form:<built-in function array>
Docstring:
array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0, ...
...
```

NumPy

In case you need help...

- Looking for something:

```
np.lookfor('create array')
Search results for 'create array'
---------------------------------
numpy.array
    Create an array.
numpy.memmap
    Create a memory-map to an array stored in a *binary* file on disk.
...
```

```
nIn [6]: p.con* ?
np.concatenate
np.conj
np.conjugate
np.convolve
```

Creating Arrays

1-D Array

```
a = np.array([0, 1, 2, 3])
a
array([0, 1, 2, 3])
a.ndim
1
a.shape
(4,)
len(a)
4
```

30 seconds challenge

Is an np.array mutable?

Creating arrays

2-D, 3-D,...

```
b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
b
array([[0, 1, 2],
       [3, 4, 5]])
b.ndim
2
b.shape
(2, 3)
len(b)     # returns the size of the first dimension
2

c = np.array([[[1], [2]], [[3], [4]]])
c
array([[[1],
        [2]],

       [[3],
        [4]]])
c.shape
(2, 2, 1)
```

Creating arrays

We almost never specify each element...

- Evenly spaced:

```
import numpy as np
a = np.arange(10) # 0 .. n-1  (!)
a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
b = np.arange(1, 9, 2) # start, end (exclusive), step
b
array([1, 3, 5, 7])
```

- or by number of points:

```
c = np.linspace(0, 1, 6)   # start, end, num-points
c
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
d = np.linspace(0, 1, 5, endpoint=False)
d
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Creating Arrays

Common arrays

```
a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
b = np.zeros((2, 2))
b
array([[ 0.,  0.],
       [ 0.,  0.]])
c = np.eye(3)
c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
d = np.diag(np.array([1, 2, 3, 4]))
d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Creating Arrays

… and… more common arrays

```
a = np.random.rand(4)       # uniform in [0, 1]
a
array([ 0.95799151,  0.14222247,  0.08777354,  0.51887998])

b = np.random.randn(4)      # Gaussian
b
array([ 0.37544699, -0.11425369, -0.47616538,  1.79664113])

np.random.seed(1234)        # Setting the random seed
```

Challenge

3 minutes challenge

Create these arrays

```
[[ 1  1  1  1]
 [ 1  1  1  1]
 [ 1  1  1  2]
 [ 1  6  1  1]]
[[0. 0. 0. 0. 0.]
 [2. 0. 0. 0. 0.]
 [0. 3. 0. 0. 0.]
 [0. 0. 4. 0. 0.]
 [0. 0. 0. 5. 0.]
 [0. 0. 0. 0. 6.]]
```

hint:Examine the docstring for diag.

2 minutes challenge

Skim through the documentation for np.tile, and use this function to construct the array:

```
[[4 3 4 3 4 3]
 [2 1 2 1 2 1]
 [4 3 4 3 4 3]
```

Basic Data Types

Check the difference

```
a = np.array([1, 2, 3])
a.dtype
dtype('int64')

b = np.array([1., 2., 3.])
b.dtype
dtype('float64')
```

Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

explicitly specify which data-type you want:

```
c = np.array([1, 2, 3], dtype=float)
c.dtype
dtype('float64')
```

Data Types

Complex

```
d = np.array([1+2j, 3+4j, 5+6*1j])
d.dtype
dtype('complex128')
```

Bool

```
e = np.array([True, False, False, True])
e.dtype
dtype('bool')
```

Strings

```
f = np.array(['Bonjour', 'Hello', 'Hallo',])
f.dtype
dtype('S7') # <--- strings containing max. 7 letters
```

and more..

int32/int64...

Array representation

Start by launching IPython in pylab mode:

```
$ ipython --pylab
```

Matplotlib is a 2D plotting package. We can import its functions as below:

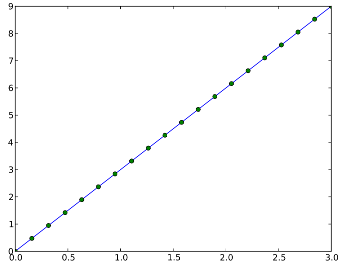
```
In [0]: import matplotlib.pyplot as plt
```

1D plotting

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 3, 20)
>>> y = np.linspace(0, 9, 20)
>>> plt.plot(x, y)       # line plot
[<matplotlib.lines.Line2D object at 0x231c450>]
>>> plt.plot(x, y, 'o')  # dot plot
[<matplotlib.lines.Line2D object at 0x225d410>]
```

this code is under CLI just to check the difference

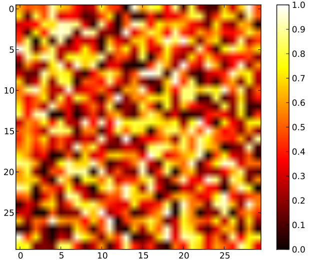
Result



Array representation

2D arrays (such as images

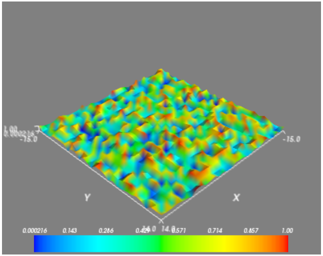
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> image = np.random.rand(30, 30)
>>> plt.imshow(image, cmap=plt.cm.hot)
<matplotlib.image.AxesImage object at 0x25f7950>
>>> plt.colorbar()
<matplotlib.colorbar.Colorbar instance at 0x2260518>
```



Array representation

3D plotting

```
In [58]: from mayavi import mlab
In [61]: mlab.surf(image)
Out[61]: <enthought.mayavi.modules.surface.Surface object at ...>
In [62]: mlab.axes()
Out[62]: <enthought.mayavi.modules.axes.Axes object at ...>
```



Challenge

5 minutes challenge

Plot a sine 1D signal with frequency $10\frac{\text{rad}}{\text{s}}$ sampled everu 0.01s during 10s.
HINT:Use np.sin

1 minute challenge

Plot the previous signal subsampled, one sample every 0.5s

Indexing and slicing

Indexing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists)

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
```

For multidimensional arrays, indexes are tuples of integers:

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
1
>>> a[2, 1] = 10 # third line, second column
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
```

Indexing and slicing

Slicing

Slicing Arrays, like other Python sequences can also be sliced:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step]
array([2, 5, 8])
```

Note that the last index is not included!

```
>>> a[:4]
array([0, 1, 2, 3])
```

All three slice components are not required: by default, start is 0, end is the last and step is 1:

```
>>> a[1:3]
array([1, 2])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[3:]
array([3, 4, 5, 6, 7, 8, 9])
```

Indexing and slicing

Small summary

```
a=np.array([np.arange(6)+10*i for i in numpy.arange(6)])
```

```
In [22]: a[4:,4:]
array([[44, 45],
       [54, 55]])

In [23]: a[:,2]
array([ 2, 12, 22, 32, 42, 52])

In [24]: a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])

In [25]: a[0,3:5]
array([3, 4])
```

```
[[ 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]]
```

Copies and views

A slicing operation creates a view on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory.

1 minute challenge

Check mutability when accessing an array from a sliced view

Use .copy() to copy a NumPy array

```
>>> a = np.arange(10)
>>> b = a[::2].copy()  # force a copy
>>> b[0] = 12
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Copies and views

Warning

This behavior can be surprising at first sight... but it allows to save both memory and time.
As a result, a matrix cannot be made symmetric in-place:

```
>>> a = np.ones((100, 100))
>>> a += a.T
>>> a
array([[ 2.,  2.,  2., ...,  2.,  2.,  2.],
       [ 2.,  2.,  2., ...,  2.,  2.,  2.],
       [ 2.,  2.,  2., ...,  2.,  2.,  2.],
       ...,
       [ 3.,  3.,  3., ...,  2.,  2.,  2.],
       [ 3.,  3.,  3., ...,  2.,  2.,  2.],
       [ 3.,  3.,  3., ...,  2.,  2.,  2.]])
```

Challenge

5 minutes challenge

Construct an array containing the prime numbers between 1 and 100

Masks

Using boolean masks

```
>>> np.random.seed(3)
>>> a = np.random.random_integers(0, 20, 15)
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
>>> (a % 3 == 0)
array([False,  True, False,  True, False, False, False,  True, False,
        True,  True, False,  True, False, False], dtype=bool)
>>> mask = (a % 3 == 0)
>>> extract_from_a = a[mask] # or,  a[a%3==0]
>>> extract_from_a           # extract a sub-array with the mask
array([ 3,  0,  9,  6,  0, 12])
```

Indexing with a mask can be very useful to assign a new value to a sub-array:

```
>>> a[a % 3 == 0] = -1
>>> a
array([10, -1,  8, -1, 19, 10, 11, -1, 10, -1, -1, 20, -1,  7, 14])
```

Indexing... more

Indexing with an array of int

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Indexing can be done with an array of integers, where the same index is repeated several time:

```
>>> a[[2, 3, 2, 4, 2]]  # note: [2, 3, 2, 4, 2] is a Python list
array([2, 3, 2, 4, 2])
```

New values can be assigned with this kind of indexing:

```
>>> a[[9, 7]] = -10
>>> a
array([  0,   1,   2,   3,   4,   5,   6, -10,   8, -10])
```

When a new array is created by indexing with an array of integers, the new array has the same shape than the array of integers:

```
>>> a = np.arange(10)
>>> idx = np.array([[3, 4], [9, 7]])
>>> a[idx]
array([[3, 4],
       [9, 7]])
>>> b = np.arange(10)
```

Challenge

1 minute challenge

Check whenever the indexing with int is a view or a copy of the original array

Fancy Indexing

Small summary

```
a=np.array([np.arange(6)+10*i for i in numpy.arange(6)])
```

```
In [53]: a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

In [54]: a[3:,[0,2,5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])
```

```
[[ 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 [55]: mask=array([1,0,1,0,0,1],dtype=bool)

In [56]: a[mask,2]
array([ 2, 22, 52])
```

Indexing

Adding axes while indexing

Indexing with the np.newaxis object allows us to add an axis to an array:

```
>>> z = np.array([1, 2, 3])
>>> z
array([1, 2, 3])
>>> z[:, np.newaxis]
array([[1],
       [2],
       [3]])
>>> z[np.newaxis, :]
array([[1, 2, 3]])
```

Numerical Operations

Elementwise operations

With scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2,  4,  8, 16])
```

All arithmetic operates elementwise:

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([ 2.,  4.,  6.,  8.])
>>> j = np.arange(5)
>>> 2**(j + 1) - j
array([ 2,  3,  6, 13, 28])
```

Numerical Operations

Warning

Array multiplication is not matrix multiplication:

```
>>> c = np.ones((3, 3))
>>> c * c                    # NOT matrix multiplication!
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Use .dot()

Note Matrix multiplication:

```
>>> c.dot(c)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

Numerical Operations

More operations

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)
```

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)
```

Numerical Operations

More operations

Shape mismatches:

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> a + np.array([1, 2])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

'Broadcast'? We'll return to that later.

Challenge

3 minutes challenge

Generate arrays [2**0, 2**1, 2**2, 2**3, 2**4]
and $a_j = 2^{3*j} - j$

Numerical Operations

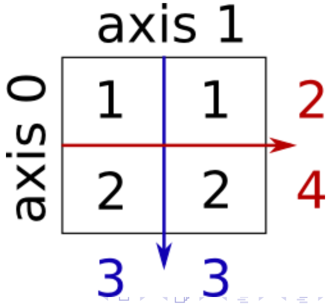
Basic reductions

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)
```



Numerical Operations

Statistics

```
>>> x = np.array([1, 2, 3, 1])
>>> y = np.array([[1, 2, 3], [5, 6, 1]])
>>> x.mean()
1.75
>>> np.median(x)
1.5
>>> np.median(y, axis=-1) # last axis
array([ 2.,  5.])
```

```
>>> x.std()   # full population standard dev.
0.82915619758884995
```

Logical operations

```
>>> np.all([True, True, False])
False
>>> np.any([True, True, False])
True
```

Extrema

```
>>> x = np.array([1, 3, 2])
>>> x.min()
1
>>> x.max()
3
```

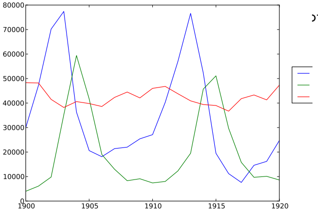
```
>>> x.argmin()  # index of minimum
0
>>> x.argmax()  # index of maximum
1
```

Note: very usefull

```
>>> a = np.zeros((100, 100))
>>> np.any(a != 0)
False
>>> np.all(a == a)
True
```

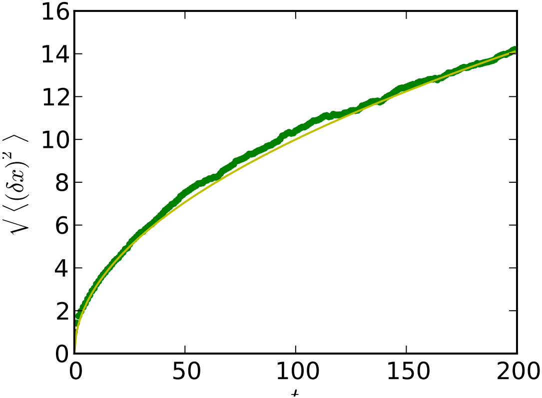
Guided exemple

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> # We can first plot the data:
>>> data = np.loadtxt('challenges/populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables
>>> plt.plot(year, hares, year, lynxes, year, carrots)
[<matplotlib.lines.Line2D object at 0x29f6d10>, <matplotlib.lin                    otlib.l
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
<matplotlib.legend.Legend object at 0x29fac90>
>>> # The mean populations over time:
>>> populations = data[:,1:]
>>> print populations.mean(axis=0)
[ 34080.95238095  20166.66666667  42400.        ]
>>> # [ 34080.95238095,  20166.66666667,  42400.        ]
>>> # The sample standard deviations:
>>> print populations.std(axis=0, ddof=1)
[ 21413.98185877  16655.99991995   3404.55577132]
>>> # [ 21413.98185877,  16655.99991995,   3404.55577132]
>>> # Which species has the highest population each year?
>>> print np.argmax(populations, axis=1)
[2 2 0 0 1 1 2 2 2 2 2 2 0 0 0 1 2 2 2 2 2]
>>> # [2, 2, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 1, 2, 2, 2, 2, 2]
```



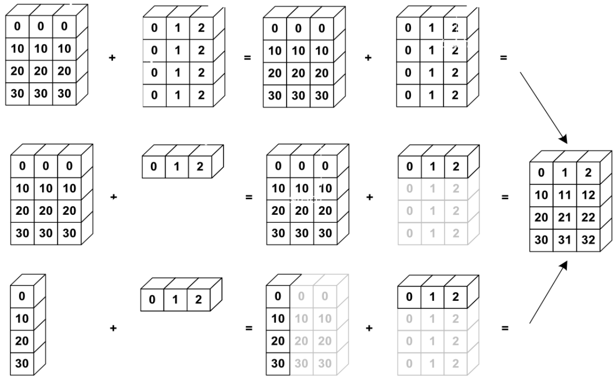
Guided exemple

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> n_stories = 1000 # number of walkers
>>> t_max = 200       # time during which we follow the walker
>>> # We randomly choose all the steps 1 or -1 of the walk
>>> t = np.arange(t_max)
>>> steps = 2 * np.random.random_integers(0, 1, (n_stories, t_max)) - 1
>>> print np.unique(steps) # Verification: all steps are 1 or -1
[-1  1]
>>> # [-1,  1]
>>> # We build the walks by summing steps along the time
>>> positions = np.cumsum(steps, axis=1) # axis = 1: dimension of time
>>> sq_distance = positions**2
>>> # We get the mean in the axis of the stories
>>> mean_sq_distance = np.mean(sq_distance, axis=0)
>>> # Plot the results:
>>> plt.figure(figsize=(4, 3))
<matplotlib.figure.Figure object at 0x24a2a90>
>>> plt.plot(t, np.sqrt(mean_sq_distance), 'g.', t, np.sqrt(t), 'y-')
[<matplotlib.lines.Line2D object at 0x2499650>, <matplotlib.lines.Line2D object at 0x248af90>]
>>> plt.xlabel(r"$t$")
<matplotlib.text.Text object at 0x26017d0>
>>> plt.ylabel(r"$\sqrt{\langle (\delta x)^2 \rangle}$")
<matplotlib.text.Text object at 0x279cf90>
```


16
14
12
10
8
6
4
2
0
0
50
100
150
200
t
$\sqrt{\langle(\delta x)^2\rangle}$

Numerical Operations

Broadcasting



Challenge

1 minute challenge

Verify that broadcasting works as specified

Numerical Operations

Broadcasting example

Let's construct an array of distances (in miles) between cities of Europe.

```
>>> mileposts = np.array([0, 198, 303, 736, 871, 1175, 1475, 1544,1913, 2448])
>>> distance_array = np.abs(mileposts - mileposts[:, np.newaxis])
>>> distance_array
array([[   0,  198,  303,  736,  871, 1175, 1475, 1544, 1913, 2448],
       [ 198,    0,  105,  538,  673,  977, 1277, 1346, 1715, 2250],
       [ 303,  105,    0,  433,  568,  872, 1172, 1241, 1610, 2145],
       [ 736,  538,  433,    0,  135,  439,  739,  808, 1177, 1712],
       [ 871,  673,  568,  135,    0,  304,  604,  673, 1042, 1577],
       [1175,  977,  872,  439,  304,    0,  300,  369,  738, 1273],
       [1475, 1277, 1172,  739,  604,  300,    0,   69,  438,  973],
       [1544, 1346, 1241,  808,  673,  369,   69,    0,  369,  904],
       [1913, 1715, 1610, 1177, 1042,  738,  438,  369,    0,  535],
       [2448, 2250, 2145, 1712, 1577, 1273,  973,  904,  535,    0]])
```

Numerical Operations

Array shape manipulation

Flattening

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.ravel()
array([1, 2, 3, 4, 5, 6])
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.ravel()
array([1, 4, 2, 5, 3, 6])
```

Reshaping

```
>>> a.shape
(2, 3)
>>> b = a.ravel()
>>> b.reshape((2, 3))
array([[1, 2, 3],
       [4, 5, 6]])
```

Example of use for Phisicists

Block matrices and vectors (and tensors)

Vector space: quantum level $\otimes$ spin

$$\check{\psi} = \begin{pmatrix} \hat{\psi}_1 \\ \hat{\psi}_2 \end{pmatrix}, \quad \hat{\psi}_1 = \begin{pmatrix} \psi_{1\uparrow} \\ \psi_{1\downarrow} \end{pmatrix}, \quad \hat{\psi}_2 = \begin{pmatrix} \psi_{2\uparrow} \\ \psi_{2\downarrow} \end{pmatrix}$$

In short: for block matrices and vectors, it can be useful to preserve the block structure.

```
>>> psi = np.zeros((2, 2))   # dimensions: level, spin
>>> psi[0, 1] # <-- psi_{1,downarrow}
0.0
```

Example of use for Phisicists

Block matrices and vectors (and tensors)

Linear operators on such block vectors have similar block structure:

$$\check{H} = \begin{pmatrix} \hat{h}_{11} & \hat{V} \\ \hat{V}^\dagger & \hat{h}_{22} \end{pmatrix}, \quad \check{h}_{11} = \begin{pmatrix} \epsilon_{1\uparrow} & 0 \\ 0 & \epsilon_{1\downarrow} \end{pmatrix}$$

```
>>> H = np.zeros((2, 2, 2, 2)) # dimensions: level1, level2, spin1, spin2
>>> h_11 = H[0,0,:,:]
>>> V = H[0,1]
```

Doing the matrix product: get rid of the block structure, do the 4x4 matrix product, then put it back

$$\check{H}\check{\psi}$$

```
>>> def mdot(operator, psi):
...     return operator.transpose(0, 2, 1, 3).reshape(4, 4).dot(
...                psi.reshape(4)).reshape(2, 2)
```

Numerical Operations

Sorting data

Sorting along an axis:

```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = np.sort(a, axis=1)
>>> b
array([[3, 4, 5],
       [1, 1, 2]])
```

In-place sort:

```
>>> a.sort(axis=1)
>>> a
array([[3, 4, 5],
       [1, 1, 2]])
```

Numerical Operations

Getting indices of the sort

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a)
>>> j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```

Also getting indices of min and max

```
>>> a = np.array([4, 3, 1, 2])
>>> j_max = np.argmax(a)
>>> j_min = np.argmin(a)
>>> j_max, j_min
(0, 2)
```

Challenge

5 minutes multy-challenge

1. Form the 2-D array (without typing it in explicitly):

```
1  6 11
2  7 12
3  8 13
4  9 14
5 10 15
```

   and generate a new array containing its 2nd and 4th rows.

2. Divide each column of the array:

```
>>> a = np.arange(25).reshape(5, 5)
```

   elementwise with the array b = np.array([1., 5, 10, 15, 20]). (Hint: np.newaxis).

Challenge

Worked challenge

From Lena picture get these modifications

Lena picture

```
>>> from scipy import misc
>>> lena = misc.lena()
```

Solution to Lena

First image

```
In [3]: import pylab as plt
In [4]: lena = misc.lena()
In [5]: plt.imshow(lena)
```

Solution to Lena

First image

```
In [3]: import pylab as plt
In [4]: lena = misc.lena()
In [5]: plt.imshow(lena)
```

Second image

```
In [6]: plt.imshow(lena, cmap=plt.cm.gray)
```

Solution to Lena

First image

```
In [3]: import pylab as plt
In [4]: lena = misc.lena()
In [5]: plt.imshow(lena)
```

Second image

```
In [6]: plt.imshow(lena, cmap=plt.cm.gray)
```

Third image

```
In [9]: crop_lena = lena[30:-30,30:-30]
```

Solution to Lena

Last image

```
In [15]: y, x = np.ogrid[0:512,0:512] # x and y indices of pixels
In [16]: y.shape, x.shape
Out[16]: ((512, 1), (1, 512))
In [17]: centerx, centery = (256, 256) # center of the image
In [18]: mask = ((y - centery)**2 + (x - centerx)**2) > 230**2 # circle
In [19]: lena[mask] = 0
In [20]: plt.imshow(lena)
```

More elaborated arrays

Casting

"Bigger" type wins in mixed-type operations:

```
>>> np.array([1, 2, 3]) + 1.5
array([ 2.5,  3.5,  4.5])
```

Assignment never changes the type!

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')
>>> a[0] = 1.9     # <-- float is truncated to integer
>>> a
array([1, 2, 3])
```

More elaborated arrays

Forced casts:

```
>>> a = np.array([1.7, 1.2, 1.6])
>>> b = a.astype(int)   # <-- truncates to integer
>>> b
array([1, 1, 1])
```

Rounding:

```
>>> a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])
>>> b = np.around(a)
>>> b                       # still floating-point
array([ 1., 2., 2., 2., 4., 4.])
>>> c = np.around(a).astype(int)
>>> c
array([ 1, 2, 2, 2, 4, 4])
```

Structured data types

Sensor example

- sensor code (4 character string)
- position (float)
- value (float)

```
>>> samples = np.zeros((6,), dtype=[('sensor_code', 'S4'),
...                                 ('position', float), ('value', float)])
>>> samples.ndim
1
>>> samples.shape
(6,)
>>> samples.dtype.names
('sensor_code', 'position', 'value')
```

Structured data types

Assignment

```
>>> samples[:] = [('ALFA',   1, 0.37), ('BETA', 1, 0.11), ('TAU', 1,   0.13),
...               ('ALFA', 1.5, 0.37), ('ALFA', 3, 0.11), ('TAU', 1.2, 0.13)]
>>> samples
array([('ALFA', 1.0, 0.37), ('BETA', 1.0, 0.11), ('TAU', 1.0, 0.13),
       ('ALFA', 1.5, 0.37), ('ALFA', 3.0, 0.11), ('TAU', 1.2, 0.13)],
      dtype=[('sensor_code', '|S4'), ('position', '<f8'), ('value', '<f8')])
```

Field access works by indexing with field names:

```
>>> samples['sensor_code']
array(['ALFA', 'BETA', 'TAU', 'ALFA', 'ALFA', 'TAU'],
      dtype='|S4')
>>> samples['value']
array([ 0.37,  0.11,  0.13,  0.37,  0.11,  0.13])
>>> samples[0]
('ALFA', 1.0, 0.37)

>>> samples[0]['sensor_code'] = 'TAU'
>>> samples[0]
('TAU', 1.0, 0.37)
```

Structured Data Types

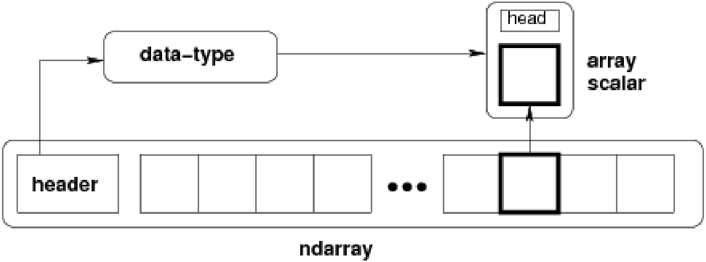
Multiple fields at once:

```
>>> samples[['position', 'value']]
array([(1.0, 0.37), (1.0, 0.11), (1.0, 0.13), (1.5, 0.37), (3.0, 0.11),
       (1.2, 0.13)],
      dtype=[('position', '<f8'), ('value', '<f8')])
```

Fancy indexing works, as usual:

```
>>> samples[samples['sensor_code'] == 'ALFA']
array([('ALFA', 1.5, 0.37), ('ALFA', 3.0, 0.11)],
      dtype=[('sensor_code', '|S4'), ('position', '<f8'), ('value', '<f8')])
```

Under the Hood



Advanced Operations

Polinomials

Numpy also contains polynomials in different bases, for instance $3x^2 + 2x - 1$

```
>>> p = np.poly1d([3, 2, -1])
>>> p(0)
-1
>>> p.roots
array([-1.        ,  0.33333333])
>>> p.order
2
```

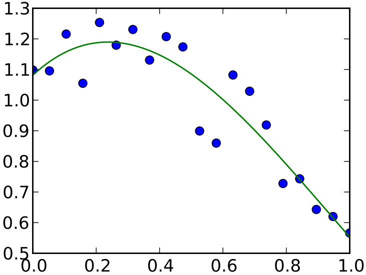
Fitting...

```
>>> x = np.linspace(0, 1, 20)
>>> y = np.cos(x) + 0.3*np.random.rand(20)
>>> p = np.poly1d(np.polyfit(x, y, 3))
```

Advanced Operations

fitting result

```
>>> t = np.linspace(0, 1, 200)
>>> plt.plot(x, y, 'o', t, p(t), '-')
```



More Polinomials

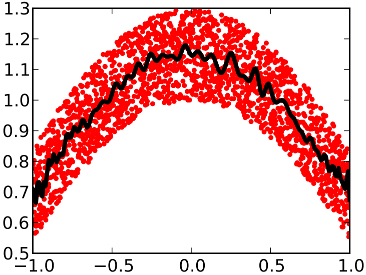
Chebyshev

$3x^2 + 2x - 1$

```
>>> p = np.polynomial.Polynomial([-1, 2, 3]) # coefs in different order!
>>> p(0)
-1.0
>>> p.roots()
array([-1.        ,  0.33333333])
>>> p.degree()  # In general polynomials do not always expose 'order'
2
```

Example

```
>>> x = np.linspace(-1, 1, 2000)
>>> y = np.cos(x) + 0.3*np.random.rand(2000)
>>> p = np.polynomial.Chebyshev.fit(x, y, 90)
>>> t = np.linspace(-1, 1, 200)
>>> plt.plot(x, y, 'r.')
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(t, p(t), 'k-', lw=3)
[<matplotlib.lines.Line2D object at ...>]
```



Outline

Matplotlib Vs Pylab

Matplotlib

matplotlib is probably the single most used Python package for 2D-graphics

Pylab

pylab provides a procedural interface to the matplotlib object-oriented plotting library. It is modeled closely after Matlab(TM). Therefore, the majority of plotting commands in pylab have Matlab(TM) analogs with similar arguments

Conclusion

Learn Pylab

How to bring pylab to work

- Fom any editor import it

```
from pylab import *
```

- start iPython with pylab

```
$ ipython --pylab
```

Process

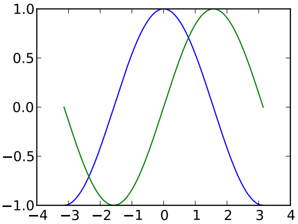
Learn by example

To show how it works lets make an evolutive graphic

Simple plot

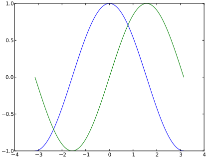
Create data and plot it

```
import pylab as pl
import numpy as np

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

pl.plot(X, C)
pl.plot(X, S)

pl.show()
```



Instantiating defaults

```
import pylab as pl
import numpy as np
# Create a figure of size 8x6 points, 80 dots per inch
pl.figure(figsize=(8, 6), dpi=80)
# Create a new subplot from a grid of 1x1
pl.subplot(1, 1, 1)
X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)
# Plot cosine with a blue continuous line of width 1 (pixels)
pl.plot(X, C, color="blue", linewidth=1.0, linestyle="-")
# Plot sine with a green continuous line of width 1 (pixels)
pl.plot(X, S, color="green", linewidth=1.0, linestyle="-")
# Set x limits
pl.xlim(-4.0, 4.0)
# Set x ticks
pl.xticks(np.linspace(-4, 4, 9, endpoint=True))
# Set y limits
pl.ylim(-1.0, 1.0)
# Set y ticks
pl.yticks(np.linspace(-1, 1, 5, endpoint=True))
# Save figure using 72 dots per inch
# savefig("exercice_2", dpi=72)
# Show result on screen
pl.show()
```

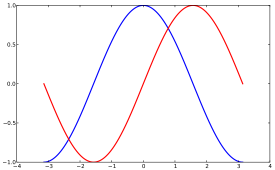


Changing colors and line widths

First step

the cosine in blue and the sine in red and a slighty thicker line for both of them. We'll also slightly alter the figure size to make it more horizontal.

```
...
pl.figure(figsize=(10, 6), dpi=80)
pl.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
pl.plot(X, S, color="red",  linewidth=2.5, linestyle="-")
...
```

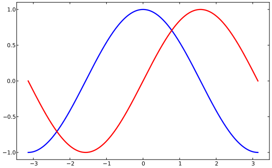


Setting limits

Second step

Lets adjust a little bit the figure limits

```
...
pl.xlim(X.min() * 1.1, X.max() * 1.1)
pl.ylim(C.min() * 1.1, C.max() * 1.1)
...
```

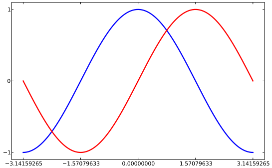


Setting tickss

Third step

Change figure tiks

```
...
pl.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])
pl.yticks([-1, 0, +1])
...
```

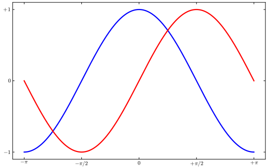


Setting tick labels

Forth step

Change figure tiks' labels

```
...
pl.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
[r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

pl.yticks([-1, 0, +1],
          [r'$-1$', r'$0$', r'$+1$'])
...
```

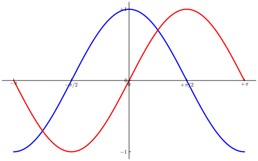


Moving spines

Forth step

Change axis positions

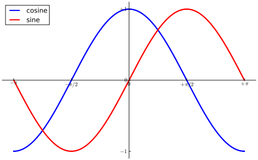
```
...
ax = pl.gca()  # gca stands for 'get current axis'
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
...
```



Adding a legend

Going on...

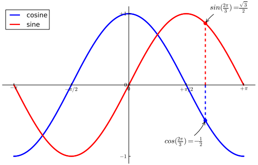
```
...
pl.plot(X, C, color="blue", linewidth=2.5, linestyle="-",
        label="cosine")
pl.plot(X, S, color="red",  linewidth=2.5, linestyle="-",
        label="sine")

pl.legend(loc='upper left')
...
```



Annotate some points

Going on...

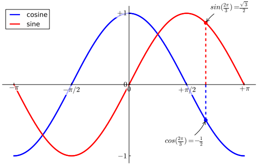
```
...
t = 2 * np.pi / 3
pl.plot([t, t], [0, np.cos(t)], color='blue', linewidth=2.5,
         linestyle="--")
pl.scatter([t, ], [np.cos(t), ], 50, color='blue')

pl.annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
            xy=(t, np.sin(t)), xycoords='data',
            xytext=(+10, +30), textcoords='offset points',
            fontsize=16,arrowprops=dict(arrowstyle="->",
            connectionstyle="arc3,rad=.2"))

pl.plot([t, t],[0, np.sin(t)], color='red', linewidth=2.5,
            linestyle="--")
pl.scatter([t, ],[np.sin(t), ], 50, color='red')

pl.annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$',
            xy=(t, np.cos(t)), xycoords='data',
            xytext=(-90, -50), textcoords='offset points',
            fontsize=16,arrowprops=dict(arrowstyle="->",
            connectionstyle="arc3,rad=.2"))
...
```



Finally...

The last detail

```
...
for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_fontsize(16)
    label.set_bbox(dict(facecolor='white', edgecolor='None',
    alpha=0.65))
...
```



Creating figures

Parameters

num	number of figure, start by 1!!!
figsize	figure.figsize,figure size in in inches (width, height)
dpi	figure.dpi ,resolution in dots per inch
facecolor	figure.facecolor, color of the drawing background
edgecolor	figure.edgecolor, color of edge around the drawing background
frameon	True, draw figure frame or not

Subplots

```
pl.figure(figsize=(6, 4))
pl.subplot(2, 2, 1)
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'subplot(2,2,1)', ha='center',
     va='center',size=20, alpha=.5)

pl.subplot(2, 2, 2)
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'subplot(2,2,2)', ha='center',
     va='center', size=20, alpha=.5)

pl.subplot(2, 2, 3)
pl.xticks(())
pl.yticks(())

pl.text(0.5, 0.5, 'subplot(2,2,3)', ha='center',
    va='center', size=20, alpha=.5)

pl.subplot(2, 2, 4)
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'subplot(2,2,4)', ha='center',
        va='center', size=20, alpha=.5)

pl.tight_layout()
pl.show()
```