Python Crash-Course

C. Basso

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova

November 20, 2007

Scientific Libraries for Python

There are three main libraries for scientific computing:

- Numpy: for vectors/matrices/tensors, linear algebra, random distributions, and something more
- Scipy: for more elaborate tasks (e.g. optimization, image processing)
- Pylab: for plotting

together they offer more or less the same functionalities of matlab
note that Numpy and Scipy can use BLAS implementations, or other numerical libraries implemented in C or Fortran; therefore they can be as fast as C

Numpy's ndarray

The most important thing of Numpy is the ndarray object:

```
>>> import numpy as NP
>>> a = NP.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.__class__
<type 'numpy.ndarray'>
>>> a + a
array([0, 2, 4, 6])
>>> a - a
array([0, 0, 0, 0])
>>> a * a
array([0, 1, 4, 9])
>>> a / a
Warning: divide by zero encountered in divide
array([0, 1, 1, 1])
```

Numpy's ndarray - shape

The array is N-dimensional:

```
>>> a.shape
(4,)
>>> a.shape = (2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> a.shape = (1,2,2)
>>> a
array([[[0, 1],
        [2, 3]]])
>>> a.shape = (1,-1)
>>> a
array([[0, 1, 2, 3]])
```

The shape can also be changed on copy:

```
>>> NP.reshape(a, (2,2))
array([[0, 1],
       [2, 3]])
>>> a.shape
(1, 4)
>>> a
array([[0, 1, 2, 3]])
```

Numpy's ndarray - type

And of course you can have different numerical types:

```
>>> a.dtype
dtype('int32')
>>> a.astype(NP.float32)
array([[ 0.,  1.,  2.,  3.]], dtype=float32)
>>> a.astype(NP.complex64)
array([[ 0.+0.j,  1.+0.j,  2.+0.j,  3.+0.j]],
      dtype=complex64)
```

But in general you could also have heterogeneous arrays, with objects of different types.

Numpy's ndarray - indexing

One of the nicest features is how the indexing is handled.

Easy:

```
>>> a.shape
(1, 4)
>>> a[0]
array([0, 1, 2, 3])
>>> a[:,0]
array([0])
>>> a[:,:2]
array([[0, 1]])
>>> a[:,::2]
array([[0, 2]])
>>> a[0,NP.newaxis].shape
(1, 4)
```

Not so easy, but useful for vectorizing computations:

```
>>> a.shape = (2,1,2)
>>> b=NP.reshape(a, (1,2,2))
>>> (a-b).shape
(2, 2, 2)
>>> a-b
array([[[ 0,  0],
        [-2, -2]],

       [[ 2,  2],
        [ 0,  0]]])
```

Initializing arrays

Four possible choices: `arange`, `zeros`, `ones`, `empty`.

```
>>> NP.arange(4)
array([0, 1, 2, 3])
>>> NP.arange(1,8,2)
array([1, 3, 5, 7])
>>> NP.zeros((1,4))
array([[ 0.,  0.,  0.,  0.]])
>>> NP.ones((1,4))
array([[ 1.,  1.,  1.,  1.]])
>>> NP.empty((1,4))
array([[  2.90522444e-310,   6.50111339e-319,
          6.63053723e+091,   8.89888303e-307]])
```

Miscellaneous methods

An incomplete list of interesting method of the ndarray class:

- `copy()`: returns a copy
- `fill(val)`: fills the array with `val`
- `transpose()`: returns the transpose
- `swapaxes(ax1, ax2)`: swap axes (dimensions) `ax1` and `ax2`
- `flatten()`: returns a 1-D copy
- `sort()` and `argsort()`, `max()` and `argmax()`, `min()` and `argmin()`
- `mean()`, `var()` and `std()`
- `...`

Special functions: `where` and `nonzero`

`nonzero(condition)`

Returns the n-dimensional indices for elements of the n-dimensional array `condition` that are nonzero into an n-tuple of equal-length index arrays.

`where(condition[,x,y])`

Returns an array shaped like `condition`, that has the elements of `x` and `y` respectively where `condition` is respectively true or false. If `x` and `y` are not given, then it is equivalent to `nonzero(condition)`.

```
>>> a
array([[0, 1, 2, 3]])
>>> NP.where(a > 0, 1, 0)
array([[0, 1, 1, 1]])
>>> NP.where(a > 0)
(array([0, 0, 0]), array([1, 2, 3]))
```

Special functions: `indices` and `mgrid`

`indices(dimensions,dtype=intp)`

Return an array of `dtype` representing n(=len(`dimensions`)) grids of indices each with variation in a single direction. The returned array has shape (n,)+dimensions.

`mgrid[index expression]`

This is an instance of a class. It can be used to construct a filled mesh-grid using slicing syntax.

```
>>> NP.indices((5,))
array([[0, 1, 2, 3, 4]])
>>> NP.indices((2,2))
array([[[0, 0],
        [1, 1]],

       [[0, 1],
        [0, 1]]])
```

```
>>> NP.mgrid[0:5:1]
array([0, 1, 2, 3, 4])
>>> NP.mgrid[0:2:1,0:2:1]
array([[[0, 0],
        [1, 1]],

       [[0, 1],
        [0, 1]]])
```

Working with matrices

Some useful methods: `dot`, `identity`, `concatenate`, `repeat`.

```
>>> a
array([[0, 1],
       [2, 3]])
>>> NP.dot(a, a)
array([[ 2,  3],
       [ 6, 11]])
>>> NP.concatenate((a, NP.identity(2)), 1)
array([[ 0.,  1.,  1.,  0.],
       [ 2.,  3.,  0.,  1.]])
>>> NP.concatenate((a,)*2, 1)
array([[0, 1, 0, 1],
       [2, 3, 2, 3]])
>>> NP.repeat(a, 2, 1)
array([[0, 0, 1, 1],
       [2, 2, 3, 3]])
```

Example: Power method

```
## Get the first eigenvalues with Power iterations.
def eig_power(A, tol=1.e-6):
    """
    Computes the eigenvector with largest eigenvalue
    of A. Returns the pair (eigval, eigvec).
    """
    assert A.ndim == 2
    b = NP.ones((A.shape[1], 1), NP.float32)
    while True:
        Ab = NP.dot(A, b)
        b_old = b.copy()
        b = Ab/norm(Ab)
        if (norm(b-b_old) / norm(b_old)) < tol:
            break
    eigval = NP.dot(NP.transpose(b), NP.dot(A, b))
    eigval /= norm2(b)
    return eigval, b
```

Some more linear algebra

The module `numpy.linalg` has the most commonly used linear algebra routines. An quick list:

- `inv`: inverse
- `pinv`: pseudo-inverse
- `det`: determinant
- `eig*`: some eigendecomposition routines
- `svd`, `qr` and `cholesky`: singular value, QR and Cholesky decompositions

Example: principal angle

```
import numpy.linalg as LA
## Returns the principal angle between two
## linear subspaces.
def principal_angle(A, B):
    "A and B must be column-orthogonal."
    svd = LA.svd(NP.dot(NP.transpose(A), B))
    return NP.arccos(min(svd[1].min(), 1.0))

>>> A = NP.arange(4).reshape((2, 2))
>>> Q, R = LA.qr(A)
>>> principal_angle(Q, Q)
0.0
>>> a = NP.array([[1, 0, 0]])
>>> b = NP.array([[0, 1, 0]])
>>> principal_angle(a, b) / NP.pi
0.5
```

Numpy vs. C: matrix multiplication

```
int main(int argc, char* argv[]) {
  int N = 1000, M = 2000;
  float matrix[N][M];
  float vector_in[M];
  float vector_out[N];
  int i, j, reps;
  for (reps = 0; reps < 100; ++reps) {
    for (i = 0; i < N; ++i) {
      vector_out[i] = 0.;
      for (j = 0; j < M; ++j) {
        vector_out[i] += matrix[i][j] * vector_in[j];
      }
    }
  }
  return 0;
}

// best out of 5 runs with time: 2.825s
```

Numpy vs. C: matrix multiplication

```
import numpy as NP

N, M = 1000, 2000
matrix = NP.empty((N, M), NP.float32)
vector_in = NP.empty((M,), NP.float32)
vector_out = NP.empty((N,), NP.float32)
for reps in xrange(100):
    vector_out = NP.dot(matrix, vector_in)

# best out of 5 runs with time: 2.357s
```

Scipy

Scipy is an other scientific package, can be thought as completing Numpy. There is some overlap because development began before Numpy was created. It works with Numpy's `ndarray`.
Some interesting modules:

- `ndimage`: image processing
- `stats`: statistical functions
- `sparse` and `umfpack`: sparse matrices
- `optimize`: optimization routines
- `linalg`: more developed module for linear algebra

scipy.ndimage

Collects functions about image processing.
Quick list:

- morphological operations (closing, opening, erosion, dilation)
- convolutions with user defined filters
- ready-to-use filters (Gaussian, Laplace, Sobel, ...)
- distance transform
- histogram
- some transformations (rotation, affine, ...)
- watershed

Examples using `scipy.ndimage`

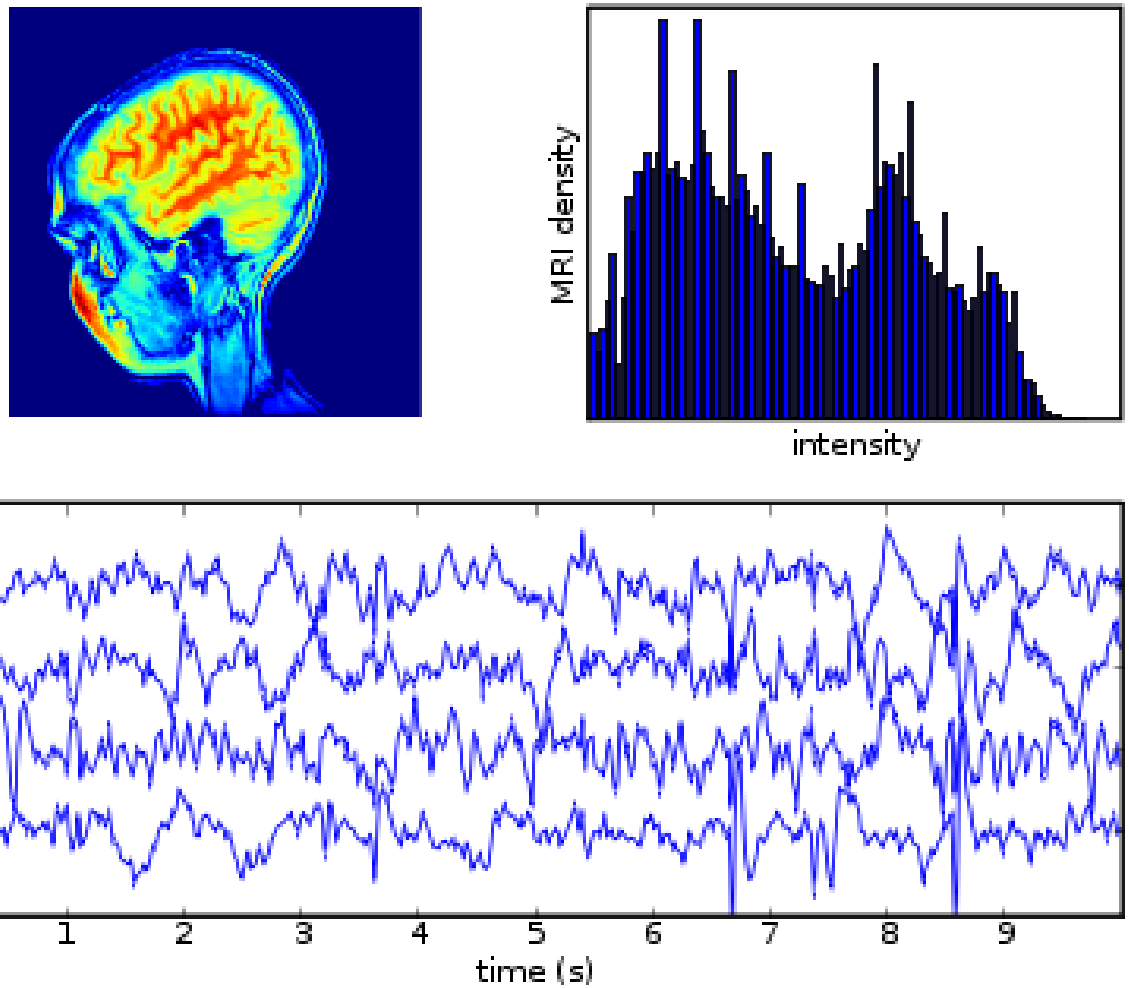
```
import scipy.ndimage as ND
import numpy as NP
## Image structure tensor.
def structure(img, size=3):
    assert img.ndim == 2
    Dx = ND.sobel(img, 1)
    Dx2 = NP.power(Dx, 2)
    Dxy = ND.sobel(Dx, 0)
    Dy2 = NP.power(ND.sobel(img, 0), 2)
    return ND.uniform_filter(Dx2, size), \
           ND.uniform_filter(Dxy, size), \
           ND.uniform_filter(Dy2, size)
```

Examples using `scipy.ndimage`

```
import scipy.ndimage as ND
import numpy as NP
## Simple KR corners detector.
def corner(img, win=2, t=0.):
    assert img.ndim[0] == 2
    Dx = ND.sobel(img, 1)
    Dy = ND.sobel(img, 0)
    Dxx = ND.sobel(Dx, 1)
    Dxy = ND.sobel(Dx, 0)
    Dyy = ND.sobel(Dy, 0)
    Dx2 = NP.power(Dx, 1)
    Dy2 = NP.power(Dy, 1)
    KR = NP.abs((Dx2*Dyy-2*Dx*Dy*Dxy+Dy2*Dxx))
    KR /= (1.e-8+Dx2+Dy2)
    return maxlocal(KR[0], win, t), KR
```

Pylab

This is a plotting package, designed to work essentially like matlab. It works with Numpy's `ndarray`.



Pylab example

MRI image:

```
from pylab import *
# creates a subplot
# in the figure
subplot(221)
# show the MRI image
imshow(im)
# turn off the axis
axis('off')
```

MRI intensity histogram:

```
from pylab import *
subplot(222)
# convert to 1-D array:
im = ravel(im)
# ignore the background:
im = take(im, nonzero(im))
# normalize:
im = im/(2.0**15)
# plots the histogram
# with 100 bins:
hist(im, 100)
# no ticks
xticks([])
yticks([])
# set the lables
xlabel('intensity')
ylabel('MRI density')
```