NumPy:
Array computing in Python

Konrad HINSEN
Centre de Biophysique Moléculaire (Orléans)
and
Synchrotron Soleil (St Aubin)

NumPy

Basic functionality for scientific computing:

- Multidimensional arrays
- Arithmetic and mathematical functions on arrays
- Linear algebra (LAPACK)
- Fourier transforms (FFTPACK)
- Random numbers

➥ Efficient implementation that makes scientific computing possible using pure Python code.

Documentation: http://numpy.scipy.org/

Arrays

- multidimensional rectangular data container
- all elements have the same type
- compact data layout, compatible with C/Fortran
- efficient operations
- arithmetic
- flexible indexing

Why arrays?

Arrays are the most "natural" data structure for many types of scientific data:

- Matrices
- Time series
- Images
- Functions sampled on a grid
- Tables of data
- ... many more ...

Python lists can handle this, right?

Why arrays?

Python lists are nice, but...

- They are slow to process
- They use a lot of memory
- For tables, matrices, or volumetric data, you need lists of lists of lists... which becomes messy to program.

```
from random import random
from operator import add
import numpy as N
n = 1000000
l1 = [random() for i in range(n)]
l2 = [random() for i in range(n)]
a1 = N.array(l1)
a2 = N.array(l2)

%timeit l3 = map(add, l1, l2)
10 loops, best of 3: 147 ms per loop

%timeit a3 = a1+a2
100 loops, best of 3: 8 ms per loop
```

Bytes per element in a list of floats: 32
Bytes per element in an array of floats: 8

Array computing genealogy

APL

- Designed around multidimensional arrays
- Every data item is an array of rank 0 or more
- Scalars are rank 0 arrays, vectors rank 1 arrays
- Rich collection of array operators
- No domain-specific functions, no standard library

Matlab

- Designed around matrices (2D arrays)
- Every data item is an array of rank 2 or more
- Vectors come in two varieties: row and column
- Huge library of domain-specific functions
- Very basic language

Array creation

- N.array([[1, 2], [3, 4]])

```
array([[1, 2],
       [3, 4]])
```

- N.zeros((2, 3), dtype=N.float)

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

- N.arange(0, 10, 2)

```
array([0, 2, 4, 6, 8])
```

- N.arange(0., 0.5, 0.1)

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4])
```

Watch out for round-off problems!

You may prefer 0.5*N.arange(5)

Optional dtype=... everywhere:

```
dtype=N.int
dtype=N.int16
dtype=N.float32
...
```

Array creation

- N.linspace(0., 1., 6)

```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

- N.eye(3)

```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

- N.diag([1., 2., 3.])

```
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
```

Indexing

a = N.arange(6)

```
array([0, 1, 2, 3, 4, 5])
```

- a[2]

```
2
```

- a[2:4]

```
array([2, 3])
```

- a[1:-1]

```
array([1, 2, 3, 4])
```

- a[:4]

```
array([0, 1, 2, 3])
```

- a[1:4:2]

```
array([1, 3])
```

- a[::-1]

```
array([5, 4, 3, 2, 1, 0])
```

This works exactly like for lists!

Indexing

a = N.array([[1, 2], [3, 4]])

```
array([[1, 2],
       [3, 4]])
```

- a[1, 0]

```
3
```

- a[1, :] a[1]

```
array([3, 4])
```

- a[:, 1]

```
array([2, 4])
```

- a[:, :, N.newaxis]

```
array([[[1],
        [2]],

       [[3],
        [4]]])
```

Indexing

```
a = N.arange(6)**2
```

```
array([0, 1, 4, 9, 16, 25])
```

- `a[a % 2 == 0]`

```
array([0, 4, 16])
```

- `a[[3, 0, 2]]`

```
array([9, 4])
```

Watch out:

- `a[N.array([True, False, False, True, False, True])]`

```
array([ 0,  9, 25])
```

- `a[[True, False, False, True, False, True]]`

```
array([1, 0, 0, 1, 0, 1])
```

Arithmetic

a = N.array([[1, 2], [3, 4]]) a.shape = (2, 2)

```
array([[1, 2],
       [3, 4]])
```

- a + a

```
array([[2, 4],
       [6, 8]])
```

- a + 1

```
array([[2, 3],
       [4, 5]])
```

- a + N.array([10, 20]) array([10, 20]).shape = (2,)

```
array([[11, 22],
       [13, 24]])
```

- a + N.array([[10], [20]]) array([[10], [20]]).shape = (2, 1)

```
array([[11, 12],
       [23, 24]])
```

Broadcasting rules

c = a + b with a.shape==(2, 3, 1) and b.shape==(3, 2)

1) len(a.shape) > len(b.shape)
   ➥ b → b[newaxis, :, :], b.shape → (1, 3, 2)

2) Compare a.shape and b.shape element by element:
   - a.shape[i] == b.shape[i]: easy
   - a.shape[i] == 1: repeat a b.shape[i] times
   - b.shape[i] == 1: repeat b a.shape[i] times
   - otherwise : error

3) Calculate the sum element by element

4) c.shape == (2, 3, 2)

Structural operations

a = (1 + N.arange(4))**2

```
array([ 1,  4,  9, 16])
```

- N.take(a, [2, 2, 0, 1]) same as a[[2, 2, 0, 1]]

```
array([9, 9, 1, 4])
```

- N.where(a >= 2, a, -1)

```
array([-1,  4,  9, 16])
```

- N.reshape(a, (2, 2))

```
array([[ 1,  4],
       [ 9, 16]])
```

- N.resize(a, (3, 5))

```
array([[ 1,  4,  9, 16,  1],
       [ 4,  9, 16,  1,  4],
       [ 9, 16,  1,  4,  9]])
```

- N.repeat(a, [2, 0, 2, 1])

```
array([ 1,  1,  9,  9, 16])
```

Functions

arccos, arcsin, arctan, arctan2, ceil, cos, cosh, exp, fabs, floor, fmod, hypot, log, log10, sin, sinh, sqrt, tan, tanh

Constants : π, e

Three sources:

- Module `math` : only for real arguments
- Module `cmath` : real and complex
- Module `numpy` :

  real, complex, **arrays**, and more..

Array structure

a = N.arange(1, 7)

```
array([1, 2, 3, 4, 5, 6])
```

b = N.reshape(a, (3, 2))

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

c = a[::3]

```
array([1, 4])
```

Watch out:

a[0] = 42

print c

```
array([42, 4])
```

Views

A view is a new array (i.e. a new Python object) that references that storage space of the array from which it was created.

If you modify array elements in the original array or in the view, they also change on the other side!

The big question: which operations return views, and which fresh arrays with independent storage areas?

Rule of thumb: An operation creates a view if this is possible for all its allowed arguments. Otherwise it returns a fresh array.

So... how do you find out if an array is a view on another arrays storage space? Check the attribute `base`.

```
a = N.arange(10)        assert a.base is None
b = a[::2]              assert b.base is a
```

Array programming

- Array operations are fast, Python loops are slow.
  (array operation = everything from module `numpy`)
- Top priority: avoid loops
- It's better to do the work three times with array operations than once with a loop.
- This does require a change of habits.
- This does require some experience.
- NumPy's array operations are designed to make this possible.

Get started with today's exercises!

Array programming strategy

- Identify the kind of operation you want to do (applying a function, filtering, rearranging, ...)
- Go through the list of array operations and check if they do that kind of operation
- Use a mixture of thinking and trying out to get the job done.
- There is often more than one way to do it.

Other NumPy functionality

Linear algebra

Operations:

- linear equations, matrix inversion
- eigenvalues, eigenvectors
- norm, determinant
- singular value decomposition
- least squares, pseudo-inverse

Restriction: general matrices only

For more information:

pydoc numpy.linalg

Fourier transforms

Operations:

- 1d general complex FFT
- 1d real FFT
- 1d hermitian FFT
- 2d complex and real FFT
- Nd complex and real FFT

For more information:

pydoc numpy.fft

Random numbers

Operations:

- single random numbers or arrays
- uniform distribution
- normal distribution
- exponential distribution
- many more!

For more information:

pydoc numpy.random

Exercises

Remember: No loops!

Array creation

Create these two arrays::

```
[[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.]]
```

```
[[ 1  1  1  1]
 [ 1  1  1  1]
 [ 1  1  1  2]
 [ 1  6  1  1]]
```

Positive elements of an array

Write a function that takes a one-dimensional array argument and returns another one-dimensional array containing the positive elements of the input array.

An example of how your function should behave:

```
import numpy as N
x = N.arange(10)-5
print x
pos_x = positive_elements(x)
print pos_x
```

prints

```
[-5 -4 -3 -2 -1  0  1  2  3  4]
[1 2 3 4]
```

Multiplication table

Write a function that takes two one-dimensional array arguments and returns a two-dimensional array containing the products of each element of the first input array with each element of the second input array.

An example of how your function should behave:

```
import numpy as N
a = N.arange(3)
b = N.array([-1., 1., 2.])
print multiplication_table(a, b)
```

prints

```
[[-0.  0.  0.]
 [-1.  1.  2.]
 [-2.  2.  4.]]
```

Hint: have another look at the indexing options, in particular `numpy.newaxis`!

Difference arrays

Write a function that takes a one-dimensional array argument and returns another one-dimensional array containing the differences between neighbouring points in the input array

An example of how your function should behave:

```
import numpy as N
x = N.array([1., 2., -3., 0.])
print differences(x)
```

prints

```
[1.  -5.  3.]
```

Hint: the simplest solution uses little more than clever indexing.

Repeating array elements

Write a function that takes a two-dimensional array argument and returns another two-dimensional array of twice the size of the input array along each dimension. Each element of the input array is copied to four adjacent elements of the output array.

An example of how your function should behave:

```
import numpy as N
a = N.array([[1, 2], [3, 4]])
print repeat_twice(a)
```

prints

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

Fitting polynomials

Write a function that fits a set of data points(x, y) to a polynomial of a given order N,

$$P_N(x) = \sum_{i=0}^{N} a_i x^i$$

and returns the fitted coefficients a_i.

Don't forget error checking: the number of data points must be greater than the number of polynomial coefficients!

Hint: Write the fitting problem as a linear least-squares fit problem of the form

$$\min_{a_j} \sum_{j=0}^{N} (M_{ij} a_j - y_i)^2$$

where the elements of M_{ij} are powers of the x_i. Use numpy.linalg.lstsq to solve this least-squares problem.