NumPy

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 handling large data 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 np
n = 1000000
l1 = [random() for i in range(n)]
l2 = [random() for i in range(n)]
a1 = np.array(l1)
a2 = np.array(l2)

%timeit l3 = map(add, l1, l2)
10 loops, best of 3: 80.8 ms per loop

%timeit a3 = a1+a2
100 loops, best of 3: 2.69 ms per loop
```

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

Never forget:

```python
import numpy as np
```

(I won't repeat this on every slide!)

Array creation

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

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

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

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

- np.arange(0, 10, 2)

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

- np.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 0.1*np.arange(5)

Optional dtype=... everywhere:

```
dtype=np.int
dtype=np.int16
dtype=np.float32
```

...

Array creation

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

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

- np.eye(3)

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

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

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

Indexing

a = np.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 = np.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[:, :, np.newaxis]

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

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

Arithmetic

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

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

- a + np.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 + np.arange(4))**2

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

- np.take(a, [2, 2, 0, 1]) or a.take([2, 2, 0, 1])

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

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

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

- np.reshape(a, (2, 2)) or a.reshape((2, 2))

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

- np.resize(a, (3, 5)) or a.resize((3, 5))

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

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

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

Statistics

a = np.arange(6)

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

- np.mean(a), np.std(a) or a.mean(), a.std()

```
(2.5, 1.707825127659933)
```

a = np.array([0, 1, 2, np.nan, 4, 5])

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

- np.mean(a), np.std(a) or a.mean(), a.std()

```
(nan, nan)
```

- np.nanmean(a), np.nanstd(a)

```
(2.3999999999999999, 1.8547236990991409)
```

Statistics

a = a = np.reshape(np.arange(4), (2,2))

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

- np.mean(a)

  ```
  1.5
  ```

- np.mean(a, axis=0)

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

- np.mean(a, axis=1)

  ```
  array([ 0.5,  2.5])
  ```

"Fancy" indexing

a = np.arange(6)**2

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

- a[a % 2 == 0] same as a.repeat(a % 2 == 0)

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

- a[[3, 0, 2]] same as a.take([3, 0, 2])

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

these are safer

Watch out:

- a[np.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])
```

Mathematical 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..

Always use module `numpy` !

Array structure

a = np.arange(1, 7)

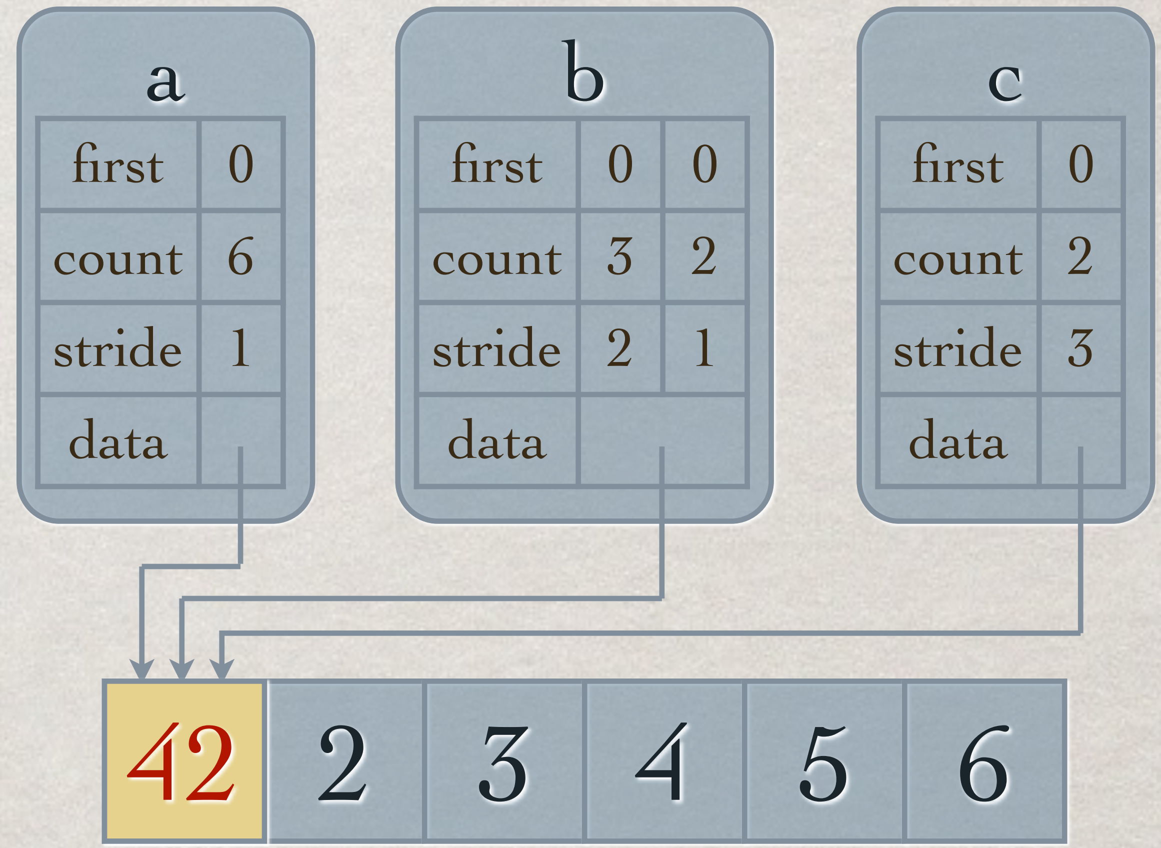
```
array([1, 2, 3, 4, 5, 6])
```

b = np.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 = np.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.

Exercises

Remember: No loops!

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 np
x = np.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]
```

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 np
x = np.array([1., 2., -3., 0.])
print differences(x)
```

prints

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

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

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 np
a = np.arange(3)
b = np.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`!

And don't "cheat" by using `numpy.outer`!