

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 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: 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

NEVER FORGET:

```
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]])`

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.]])`

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])`

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)`

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])`

Optional dtype=...
everywhere:

`dtype=np.int`

`dtype=np.int16`

`dtype=np.float32`

...

Watch out for round-off problems!

You may prefer `0.1*np.arange(5)`

ARRAY CREATION

▷ `np.linspace(0., 1., 6)`
`array([0. , 0.2, 0.4, 0.6, 0.8, 1.])`

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.]])
```

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])
```

INDEXING

```
a = np.arange(6)  
    array([0, 1, 2, 3, 4, 5])
```

```
▷ a[2]  
    2
```


INDEXING

```
a = np.arange(6)  
    array([0, 1, 2, 3, 4, 5])
```

▷ a[2]
 2

▷ a[2:4]
 array([2, 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])
```

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])
```

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])
```

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])
```

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]])
```

INDEXING

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

```
▷ a[1, 0]  
3
```


INDEXING

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

▷ `a[1, 0]`
3

▷ `a[1, :]` `a[1]`
array([3, 4])

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])

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]])
```

ARITHMETIC

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

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

▷ a + a

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

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]])
```

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

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

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]])
```


STRUCTURAL OPERATIONS

```
a = (1 + np.arange(4))**2  
array([ 1,  4,  9, 16])
```

STRUCTURAL OPERATIONS

```
a = (1 + np.arange(4))**2
```

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

```
▷ np.take(a, [2, 2, 0, 1])
```

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

same as `a[[2, 2, 0, 1]]`

STRUCTURAL OPERATIONS

```
a = (1 + np.arange(4))**2
```

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

```
▷ np.take(a, [2, 2, 0, 1])
```

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

same as a[[2, 2, 0, 1]]

```
▷ np.where(a >= 2, a, -1)
```

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

STRUCTURAL OPERATIONS

```
a = (1 + np.arange(4))**2
```

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

```
▷ np.take(a, [2, 2, 0, 1])
```

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

same as a[[2, 2, 0, 1]]

```
▷ np.where(a >= 2, a, -1)
```

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

```
▷ np.reshape(a, (2, 2))
```

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

STRUCTURAL OPERATIONS

```
a = (1 + np.arange(4))**2
```

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

```
▷ np.take(a, [2, 2, 0, 1])
```

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

same as a[[2, 2, 0, 1]]

```
▷ np.where(a >= 2, a, -1)
```

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

```
▷ np.reshape(a, (2, 2))
```

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

```
▷ np.resize(a, (3, 5))
```

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

STRUCTURAL OPERATIONS

```
a = (1 + np.arange(4))**2
```

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

▷ `np.take(a, [2, 2, 0, 1])` same as `a[[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))`

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

▷ `np.resize(a, (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])`

```
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

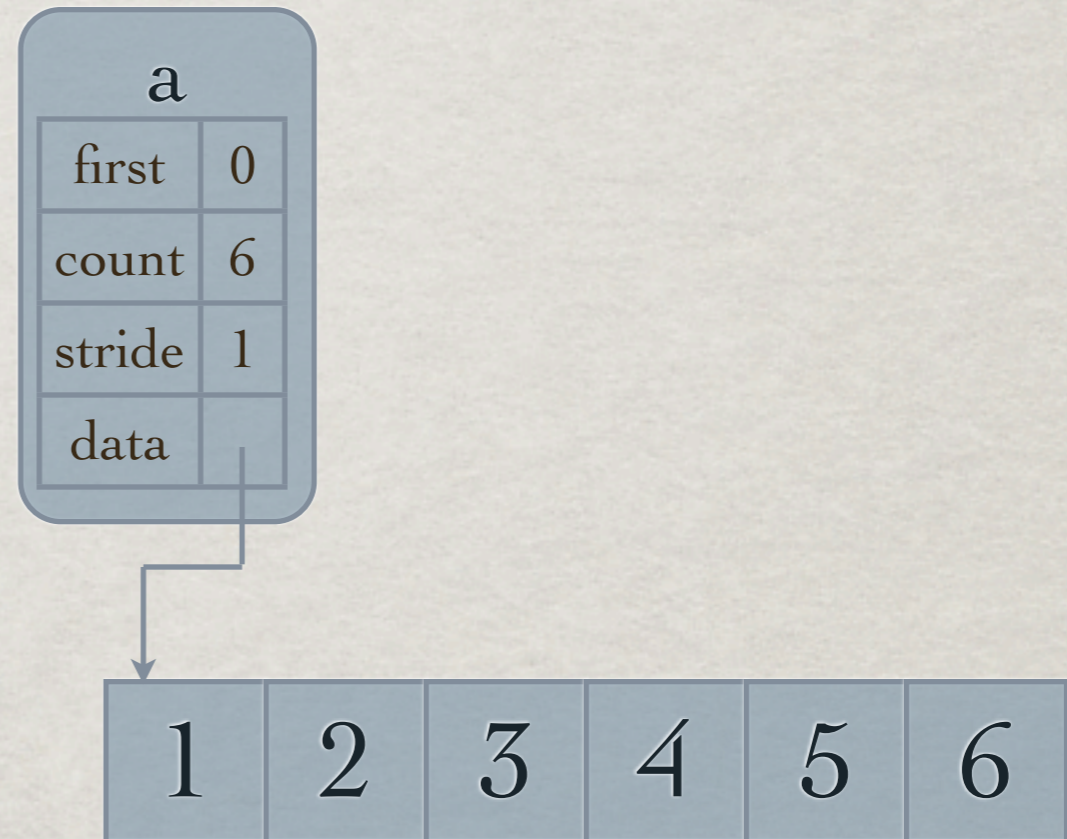
Three sources:

- ▷ Module math : only for real arguments
- ▷ Module cmath : real and complex
- ▷ Module numpy :
real, complex, **arrays**, and more..

Always use module numpy !

ARRAY STRUCTURE

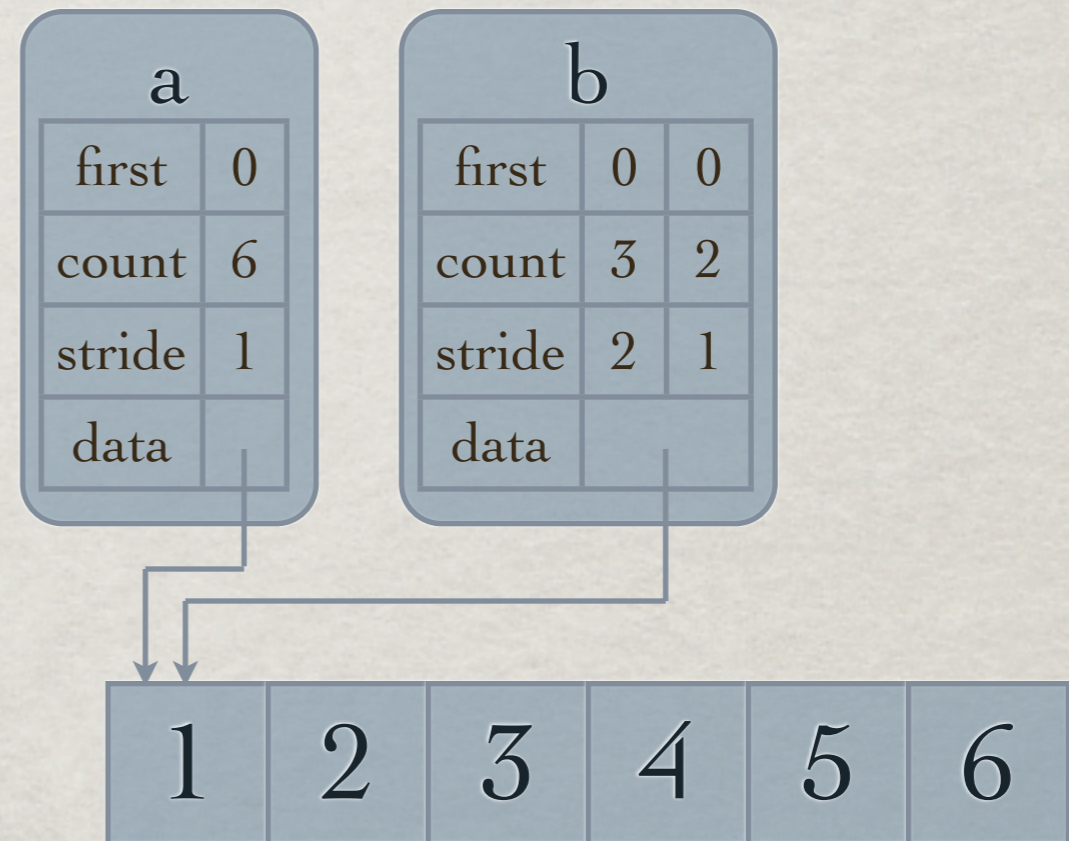
```
a = np.arange(1, 7)  
array([1, 2, 3, 4, 5, 6])
```



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]])
```

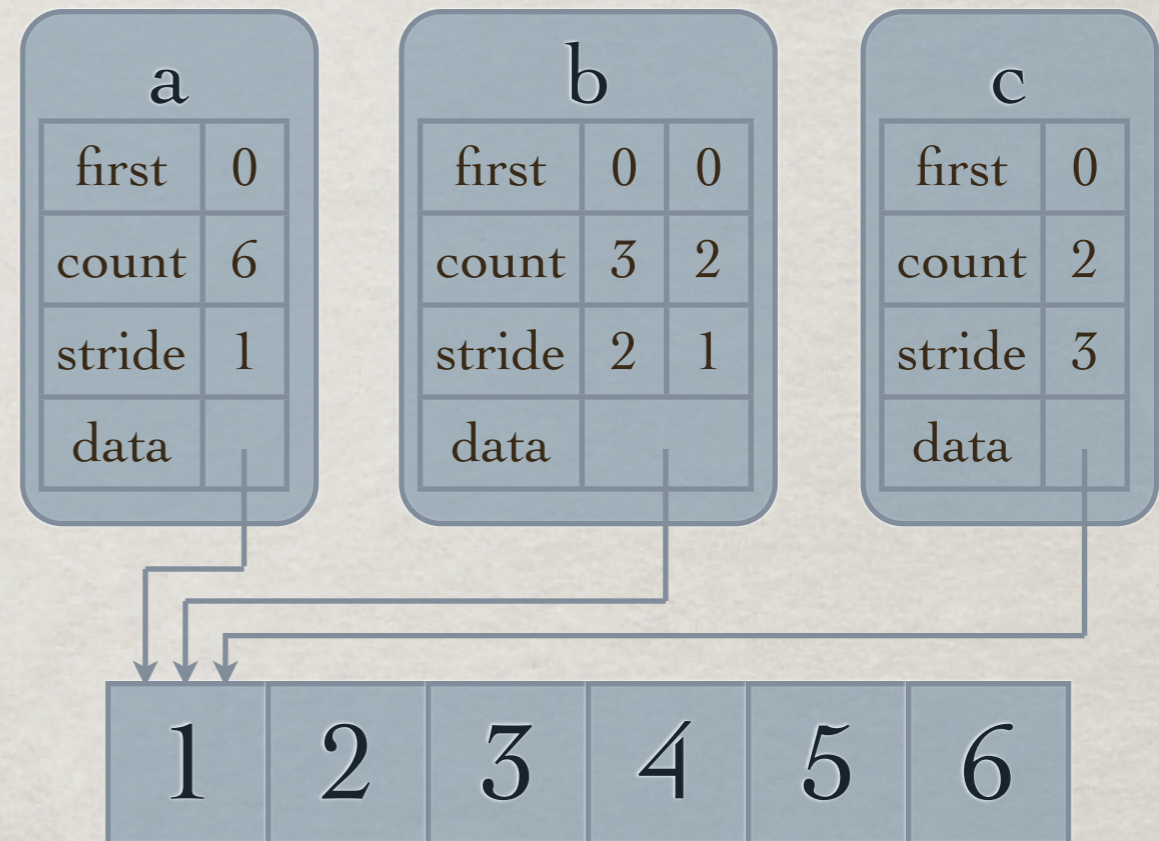


ARRAY STRUCTURE

```
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])
```

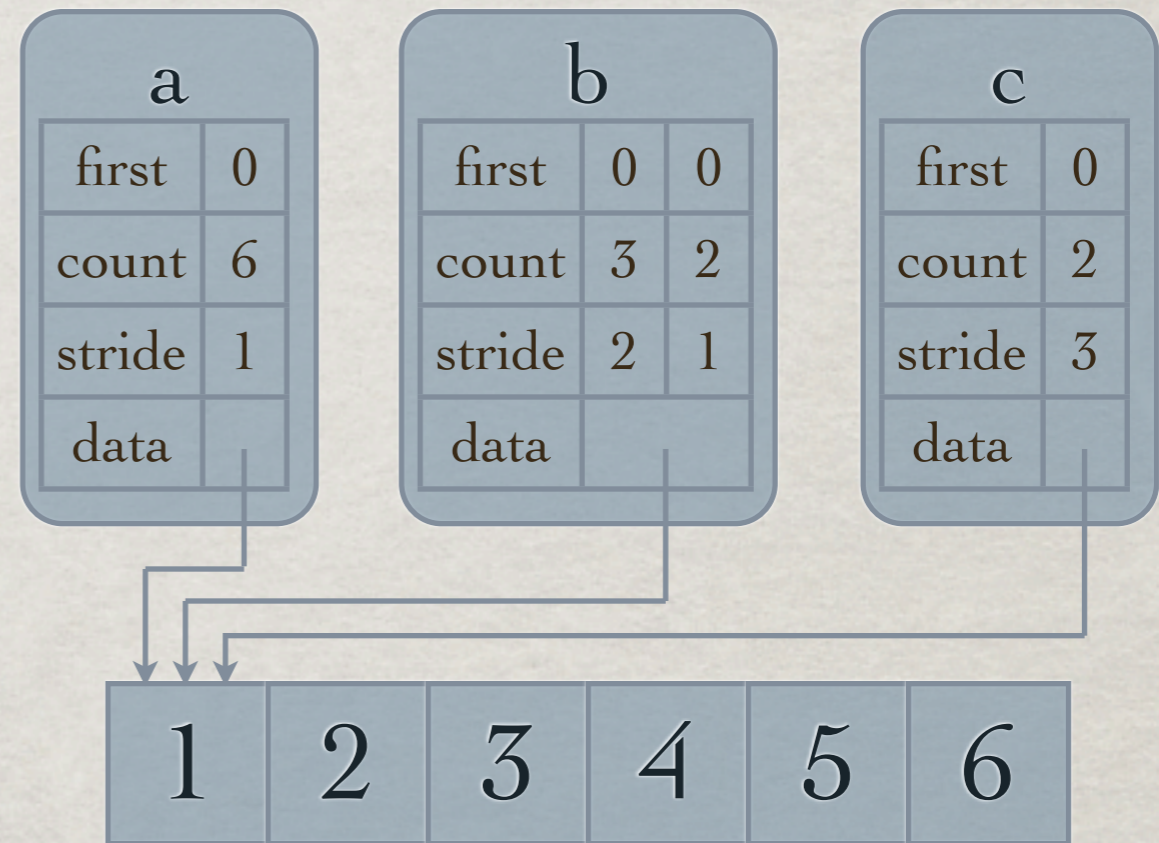


ARRAY STRUCTURE

```
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:

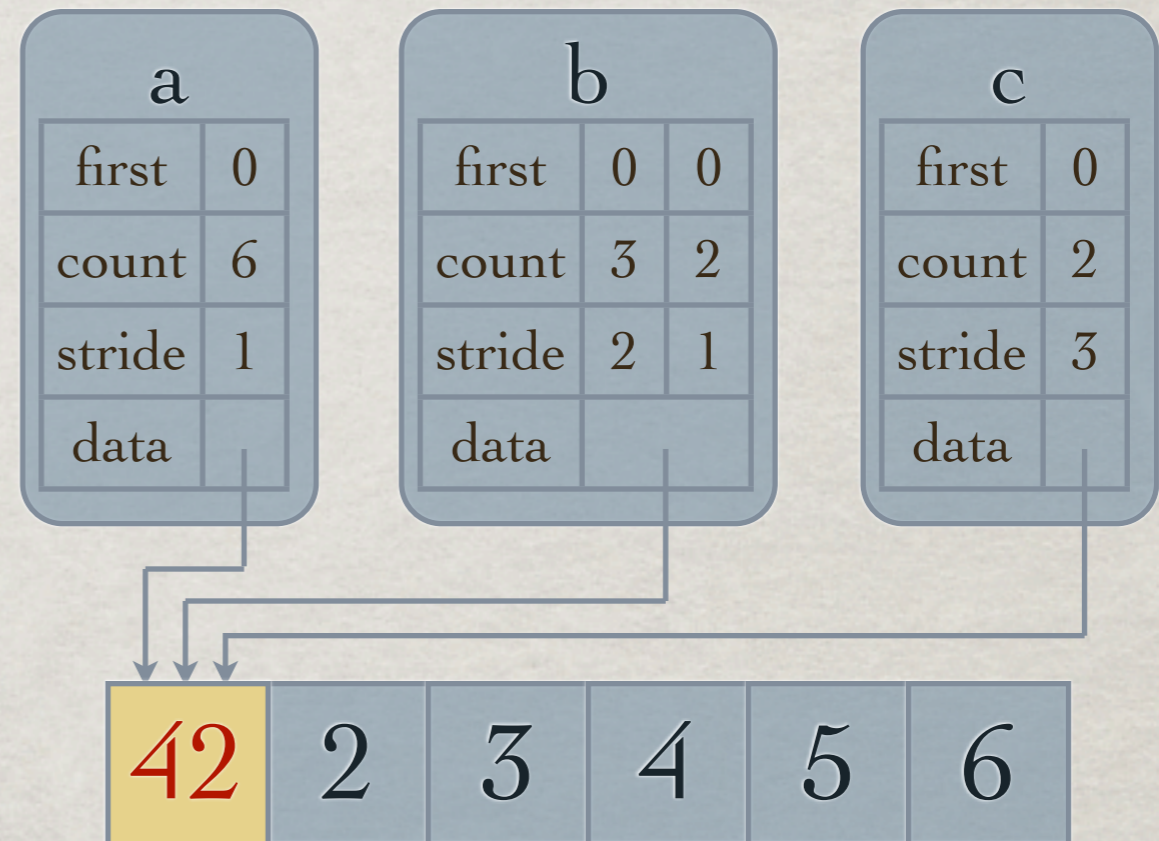
$a[0] = 42$

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:

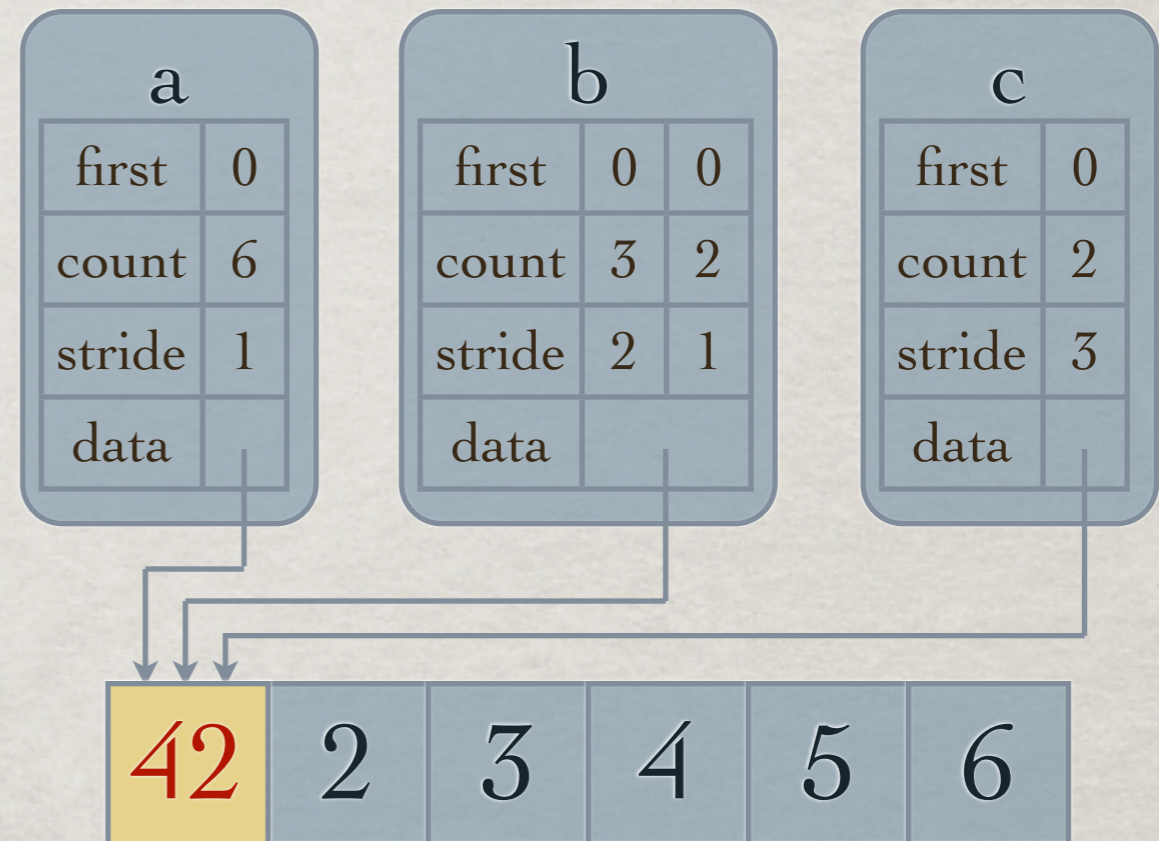
$a[0] = 42$

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])
```

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!

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 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]
```

Hint: your function should be very small:

```
def positive_elements(x):
    return ...
```

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`!

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.