The NumPy Array: A Structure for Efficient Numerical Computation

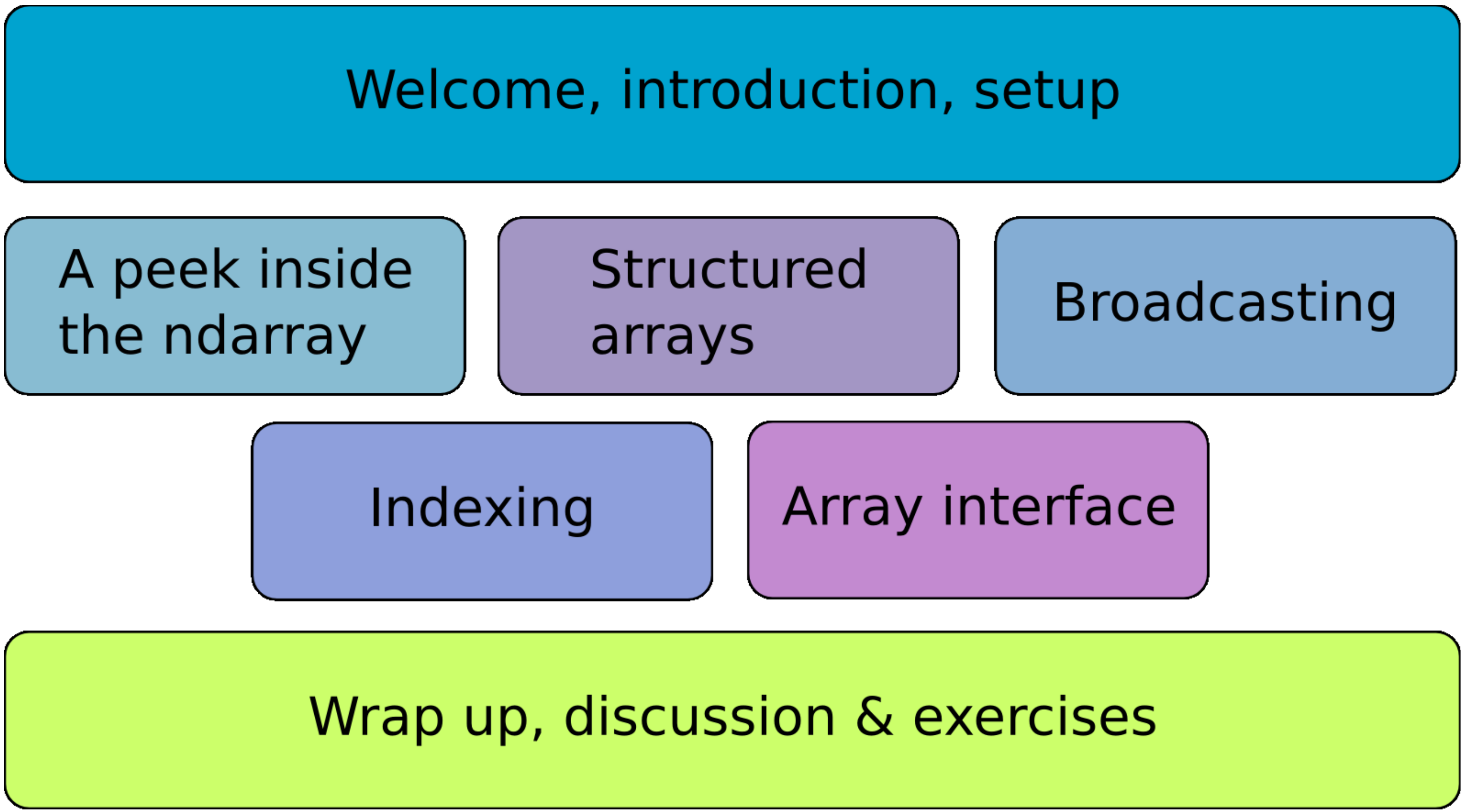
Presented at the G-Node Autumn School on

Advanced Scientific Programming in Python,

held in Kiel, Germany

Stéfan van der Walt
UC Berkeley

September, 2012


Welcome, introduction, setup
A peek inside the ndarray
Structured arrays
Broadcasting
Indexing
Array interface
Wrap up, discussion & exercises

Num-What?

This talk discusses some of the more advanced NumPy features. If you've never seen NumPy before, you may have more fun doing this tutorial:

`http://mentat.za.net/numpy/intro/intro.html`

You can always catch up by reading:

'*The NumPy array: a structure for efficient numerical computation*'. Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. In IEEE Computing in Science Engineering, March/April 2011.

Setup


```
import numpy as np    # we always use this convention,
                      # also in the documentation

print np.__version__ # version 1.5 or greater
print np.show_config() # got ATLAS/Accelerate/MKL?
```

ATLAS is a fast implementation of BLAS (Basic Linear Algebra Routines). On OSX you have Accelerate; students can get Intel's MKL for free. On Ubuntu, install `libatlas3gf-base`.

Make use of **IPython**'s powerful features! TAB-completion, documentation, source inspection, timing, cpaste, etc.

The accompanying problem sets are on the Wiki at

https://python.g-node.org/wiki/numpy

The NumPy ndarray

Revision: Structure of an ndarray

Taking a look at `numpy/core/include/numpy/ndarraytypes.h`:

```
typedef struct PyArrayObject {
  PyObject_HEAD
  char *data;               /* pointer to data buffer */
  int nd;                   /* number of dimensions */
  npy_intp *dimensions;     /* size in each dimension */
  npy_intp *strides;        /* bytes to jump to get
                             * to the next element in
                             * each dimension
                             */
  PyObject *base;           /* Pointer to original array
                            /* Decref this object */
                            /* upon deletion. */
  PyArray_Descr *descr;     /* Pointer to type struct */
  int flags;                /* Flags */
  PyObject *weakreflist;    /* For weakreferences */
} PyArrayObject;
```

A homogeneous container

```
char *data;              /* pointer to data buffer */
```

Data is just a pointer to bytes in memory:

```
In [16]: x = np.array([1, 2, 3])

In [22]: x.dtype
Out[22]: dtype('int32') # 4 bytes

In [18]: x.__array_interface__['data']
Out[18]: (26316624, False)

In [21]: str(x.data)
Out[21]: '\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
```

Dimensions


```
int nd;                   /* number of dimensions */
npy_intp *dimensions;     /* size in each dimension */
```

```
In [3]: x = np.array([])
In [4]: x.shape
Out[4]: (0,)

In [5]: np.array(0).shape
Out[5]: ()

n [8]: x = np.random.random((3, 2, 3, 3))
In [9]: x.shape
Out[9]: (3, 2, 3, 3)
In [10]: x.ndim
Out[10]: 4
```

Data type descriptors

```
PyArray_Descr *descr;  /* Pointer to type struct */
```

Common types in include int, float, bool:

```
In [19]: np.array([-1, 0, 1], dtype=int)
Out[19]: array([-1,  0,  1])
In [20]: np.array([-1, 0, 1], dtype=float)
Out[20]: array([-1.,  0.,  1.])
In [21]: np.array([-1, 0, 1], dtype=bool)
Out[21]: array([ True, False,  True], dtype=bool)
```

Each item in the array has to have the same type (occupy a fixed nr of bytes in memory), but that does not mean a type has to consist of a single item:

```
In [2]: dt = np.dtype([('value', np.int), ('status', np.bool)])
In [3]: np.array([(0, True), (1, False)], dtype=dt)
Out[3]:
 array([(0, True), (1, False)],
       dtype=[('value', '<i4'), ('status', '|b1')])
```

This is called a **structured array**.

Strides


```
npy_intp *strides;        /* bytes to jump to get */
                          /* to the next element */

In [37]: x = np.arange(12).reshape((3,4))

In [38]: x
Out[38]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [39]: x.dtype
Out[39]: dtype('int32')

In [40]: x.dtype.itemsize
Out[40]: 4

In [41]: x.strides
Out[41]: (16, 4) # (4*itemsize, itemsize)
                 # (skip_bytes_row, skip_bytes_col)
```

Flags

```c
int flags;                /* Flags */
```

```
In [66]: x = np.array([1, 2, 3]); z = x[::2]

In [67]: x.flags
Out[67]:
  C_CONTIGUOUS : True    # C-contiguous
  F_CONTIGUOUS : True    # Fortran-contiguous
  OWNDATA : True         # are we responsible for memory handling?
  WRITEABLE : True       # may we change the data?
  ALIGNED : True         # appropriate hardware alignment
  UPDATEIFCOPY : False   # update base on deallocation?

In [68]: z.flags
Out[68]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

Base Pointer

```
PyObject *base; /* Decref this object on deletion */
                /* of the array. For views, points */
                /* to original array. */
```

Trick: Deallocating foreign memory

An ndarray can be constructed from memory obtained from another library. Often, we'd like to free that memory after we're done with the array, but **numpy** can't deallocate it safely. As such, we need to trick numpy into calling the foreign library's deallocation routine. How do we do this? We assign a special object that frees the foreign memory upon object deletion to the ndarray's **base** pointer.

PyObject* **`PyCObject_FromVoidPtr`**(void* *cobj*, void (*destr)(void *))

Return value: New reference.

Create a `PyCObject` from the `void *` *cobj*. The *destr* function will be called when the object is reclaimed, unless it is *NULL*.

See Travis Oliphant's blog entry at `http://blog.enthought.com/?p=410`.

Structured arrays

Intro to structured arrays

Like we saw earlier, each item in an array has the same type, but that does not mean a type has to consist of a single item:

```
In [2]: dt = np.dtype([('value', np.int), ('status', np.bool)])
In [3]: np.array([(0, True), (1, False)], dtype=dt)
Out[3]:
 array([(0, True), (1, False)],
       dtype=[('value', '<i4'), ('status', '|b1')])
```

This is called a **structured array**, and is accessed like a dictionary:

```
In [3]: x = np.array([(0, True), (1, False)],
                                 dtype=dt)

In [5]: x['value']
Out[5]: array([0, 1])

In [6]: x['status']
Out[6]: array([ True, False], dtype=bool)
```

Structured arrays

Time	Size	Position				Gain	Samples (2048) ...			
		Az	El	Type	ID					
1172581077060	4108	0.715594	–0.148407	1	4	40	561	1467	997	–30
1172581077091	4108	0.706876	–0.148407	1	4	40	7	591	423	
1172581077123	4108	0.698157	–0.148407	1	4	40	49	–367	–565	–35
1172581077153	4108	0.689423	–0.148407	1	4	40	–55	–953	–1151	–30
1172581077184	4108	0.680683	–0.148407	1	4	40	–719	–1149	–491	38
1172581077215	4108	0.671956	–0.148407	1	4	40	–1503	–683	661	149
1172581077245	4108	0.663232	–0.148407	1	4	40	–2731	–281	2327	291
1172581077276	4108	0.654511	–0.148407	1	4	40	–3493	–159	3277	380
1172581077306	4108	0.645787	–0.148407	1	4	40	–3255	–247	3145	385
1172581077339	4108	0.637058	–0.148407	1	4	40	–2303	–101	2079	247
1172581077370	4108	0.628321	–0.148407	1	4	40	–1495	–553	571	107
1172581077402	4108	0.619599	–0.148407	1	4	40	–955	–1491	–1207	–25
1172581077432	4108	0.61087	–0.148407	1	4	40	–875	–3009	–2987	–93
1172581077463	4108	0.602148	–0.148407	1	4	40	–491	–3681	–4193	–175
1172581077497	4108	0.593438	–0.148407	1	4	40	167	–3501	–4573	–250
1172581077547	4108	0.584696	–0.148407	1	4	40	1007	–2613	–4463	–303
1172581077599	4108	0.575972	–0.148407	1	4	40	1261	–2155	–4299	–339
1172581077650	4108	0.567244	–0.148407	1	4	40	1537	–2633	–4945	–367
1172581077702	4108	0.558511	–0.148407	1	4	40	1105	–3701	–6129	–433

Reading data from file

Reading this kind of data can be somewhat troublesome:

```
while ((count > 0) && (n <= NumPoints))
  % get time - I8 [ms]
  [lw, count] = fread(fid, 1, 'uint32');
  if (count > 0) % then carry on
    uw = fread(fid, 1, 'int32');
    t(1,n) = (lw+uw*2^32)/1000;

    % get number of bytes of data
    numbytes = fread(fid, 1, 'uint32');

    % read sMEASUREMENTPOSITIONINFO (11 bytes)
    m(1,n) = fread(fid, 1, 'float32'); % az [rad]
    m(2,n) = fread(fid, 1, 'float32'); % el [rad]
    m(3,n) = fread(fid, 1, 'uint8');   % region type
    m(4,n) = fread(fid, 1, 'uint16');  % region ID
    g(1,n) = fread(fid, 1, 'uint8');

    numsamples = (numbytes-12)/2; % 2 byte integers
    a(:,n) = fread(fid, numsamples, 'int16');
```

Reading data from file

The NumPy solution:

```
dt = np.dtype([('time', np.uint64),
               ('size', np.uint32),
               ('position', [('az', np.float32),
                             ('el', np.float32),
                             ('region_type', np.uint8),
                             ('region_ID', np.uint16)]),
               ('gain', np.uint8),
               ('samples', (np.int16, 2048))])
```

```
data = np.fromfile(f, dtype=dt)
```

We can then access this structured array as before:

```
data['position']['az']
```

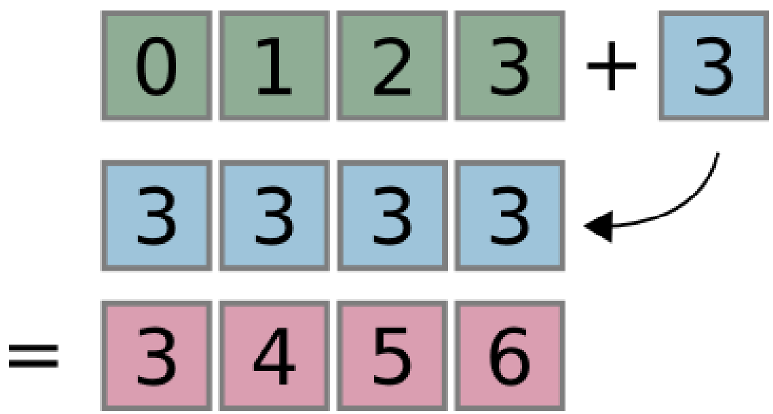
Broadcasting

Broadcasting overview (1D)

Combining of differently shaped arrays without creating large intermediate arrays:

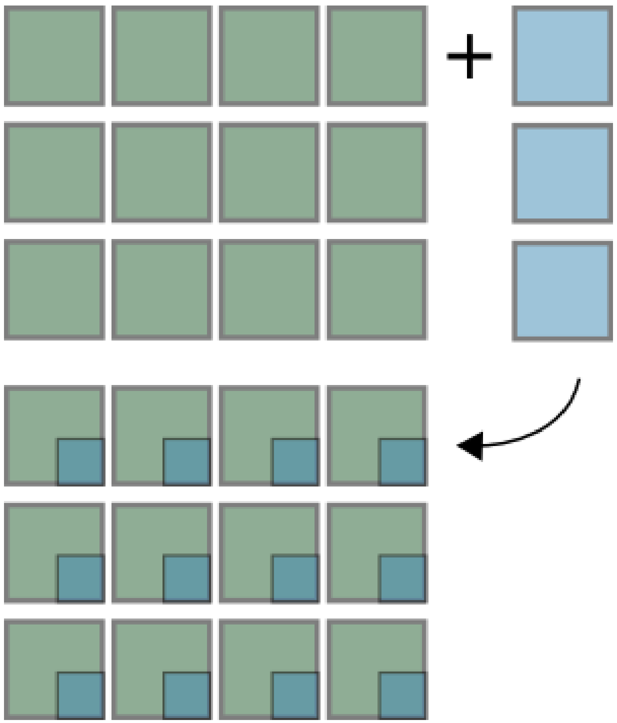
```
>>> x = np.arange(4)
>>> x = array([0, 1, 2, 3])
>>> x + 3
array([3, 4, 5, 6])
```

See the `np.doc.broadcasting` docstring for more detail.



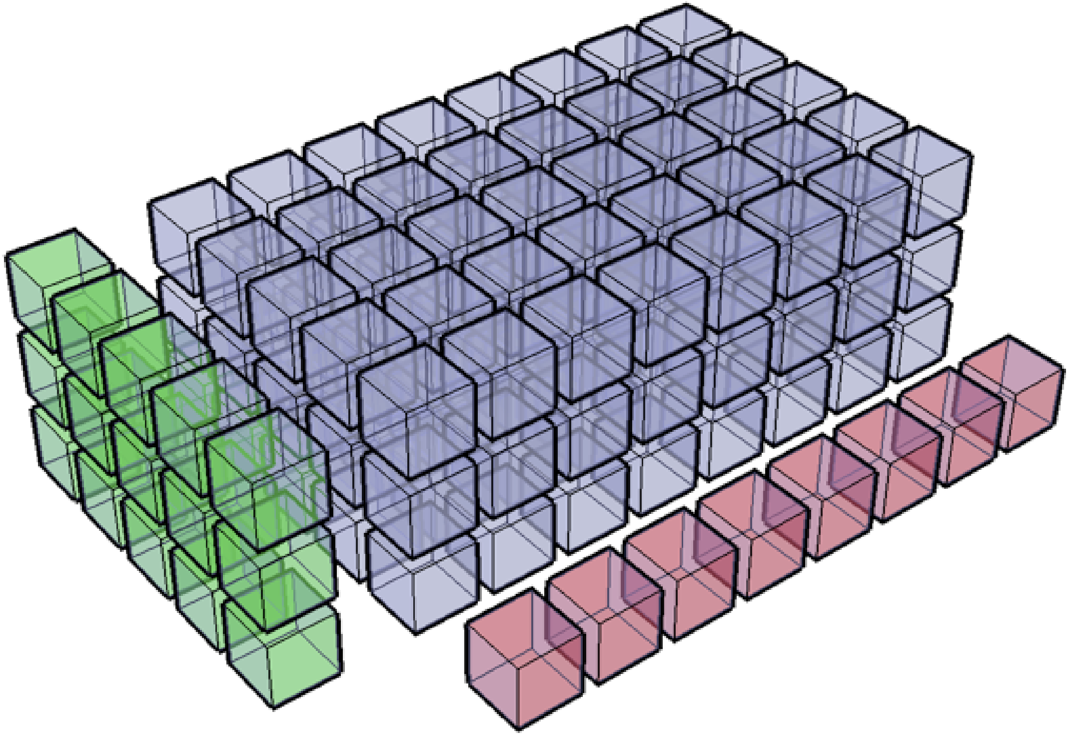
Broadcasting overview (2D)


```
In [2]: a = np.arange(12).reshape((3, 4))
In [3]: b = np.array([1, 2, 3])[:, np.newaxis]
In [4]: a + b
Out[4]:
array([[ 1,  2,  3,  4],
       [ 6,  7,  8,  9],
       [11, 12, 13, 14]])
```



Broadcasting overview (3D)


```
>>> x = np.zeros((3, 5))
>>> y = np.zeros(8)
>>> (x[..., None] + y).shape (3, 5, 8)
```



Broadcasting Rules

The broadcasting rules are straightforward—mostly. Compare dimensions, starting from the last. Match when either dimension is one or None, or if dimensions are equal:

```
Scalar    2D        3D             Bad

(  ,)     (3, 4)    (3, 5, 1)      (3, 5, 2)
(3,)      (3, 1)    (      8)      (      8)
----      ------    ----------     ---------
(3,)      (3, 4)    (3, 5, 8)         XXX
```

Explicit broadcasting


```
In [46]: xx, yy = np.broadcast_arrays(x, y)
In [47]: x = np.zeros((3, 5, 1))
In [48]: y = np.zeros((3, 5, 8))
In [49]: xx, yy = np.broadcast_arrays(x, y)
In [50]: xx.shape
Out[50]: (3, 5, 8)

In [51]: np.broadcast_arrays([1,2,3], [[1],[2],[3]])
Out[51]:
[array([[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]]),
 array([[1, 1, 1],
        [2, 2, 2],
        [3, 3, 3]])]
```

Fancy Indexing

Introduction

Remember that ndarray can be indexed in two ways:

- Using slices and scalars
- Using ndarrays («fancy indexing»)

Simple fancy indexing example:

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

>>> x[:, [1, 1, 2]]
array([[1, 1, 2],
       [4, 4, 5],
       [7, 7, 8]])

>>> np.array((x[:, 1], x[:, 1], x[:, 2])).T
array([[1, 1, 2],
       [4, 4, 5],
       [7, 7, 8]])
```

Output shape of an indexing op

1. Broadcast all index arrays against one another.
2. Use the dimensions of slices as-is.

```
>>> x = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
>>> print x
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> print x.shape
(3, 3)
>>> idx0 = np.array([[0, 1], [1, 2]]) # row indices
>>> idx1 = np.array([[0, 1]]) # column indices
```

But what would now happen when we do

```
>>> x[idx0, idx1]  ???
```

Output shape of an indexing op (cont'd)

The dimensions of idx0 and idx1 are not the same, but are they «compatible»?

```
>>> print idx0.shape, idx1.shape
(2, 2) (1, 2)
>>> a, b = np.broadcast_arrays(idx0, idx1)
>>> print a
[[0 1]
 [1 2]]
>>> print b
[[0 1]
 [0 1]]
```

Can we now predict the output? Yes.

```
>>> print x
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> print x[idx0, idx1]
[[0 4]
 [3 7]
```

Output shape of an indexing op (cont'd)


```
>>> x = np.random.random((15, 12, 16, 3))

>>> index_one = np.array([[0, 1], [2, 3], [4, 5]])
>>> index_one.shape
(3, 2)
>>> index_two = np.array([[0, 1]])
>>> index_two.shape
(1, 2)
```

Predict the output shape of:

```
x[5:10, index_one, :, index_two]
```

Warning! When mixing slicing and fancy indexing, the *order* of the output dimensions are somewhat unpredictable. Play it safe and don't mix the two!

Output shape of an indexing op (cont'd)


```
>>> x = np.random.random((15, 12, 16, 3))

>>> index_one = np.array([[0, 1], [2, 3], [4, 5]])
>>> index_one.shape
(3, 2)
>>> index_two = np.array([[0, 1]])
>>> index_two.shape
(1, 2)
```

Broadcast `index1` against `index2`:

```
(3, 2)  # shape of index_one
(1, 2)  # shape of index_two
------
(3, 2)
```

The shape of `x[5:10, index_one, :, index_two]` is

```
(3, 2, 5, 16)
```

Jack's Dilemma

Indexing and broadcasting are intertwined, as we'll see in the following example. One of my favourites from the NumPy mailing list:

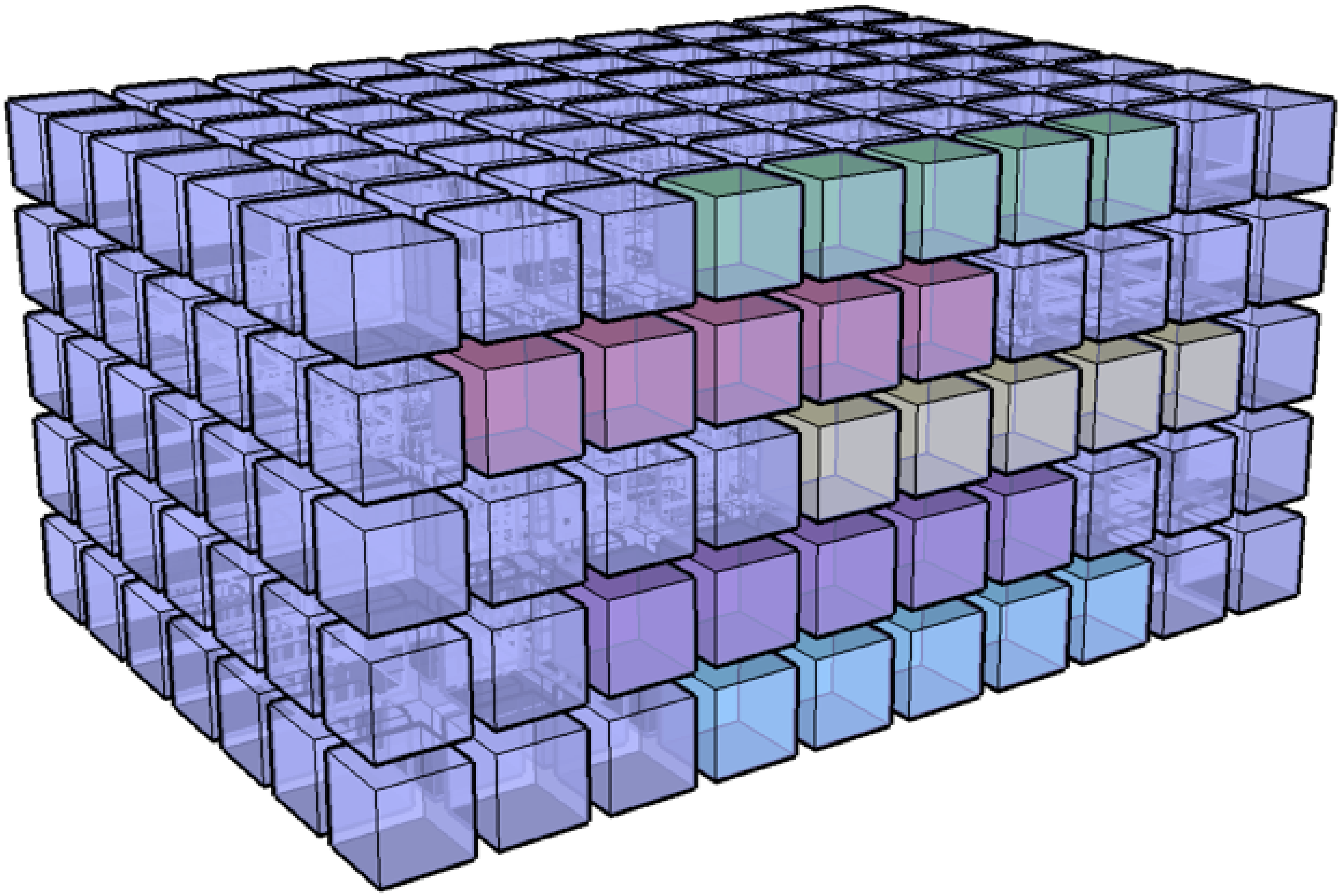
```
Date:  Wed, 16 Jul 2008 16:45:37 -0500
From:  <Jack.Cook@>
To:  <numpy-discussion@scipy.org>
Subject:  Numpy Advanced Indexing Question

Greetings,

I have an I,J,K 3D volume of amplitude values at regularly sampled
time intervals.  I have an I,J 2D slice which contains a time (K)
value at each I, J location.  What I would like to do is extract a
subvolume at a constant +/- K window around the slice.  Is there an
easy way to do this using advanced indexing or some other method?
Thanks in advanced for your help.

- Jack
```

Jack’s Dilemma (cont’d)



Test setup for Jack's problem


```
>>> ni, nj, nk = (10, 15, 20)

# Make a fake data block such that block[i,j,k] == k for all i,j,k.
>>> block = np.empty((ni, nj, nk), dtype=int)
>>> block[:,:,:] = np.arange(nk)[np.newaxis, np.newaxis, :]

# Pick out a random fake horizon in k.
>>> k = np.random.randint(5, 15, size=(ni, nj))
>>> k
array([[ 6,  9, 11, 10,  9, 10,  8, 13, 10, 12, 13,  9, 12,  5,  6],
       [ 7,  9,  6, 14, 11,  8, 12,  7, 12,  9,  7,  9,  8, 10, 13],
       [10, 14,  9, 13, 12, 11, 13,  6, 11,  9, 14, 12,  6,  8, 12],
       [ 5, 11,  8, 14, 10, 10, 10,  9, 10,  5,  7, 11,  9, 13,  8],
       [ 7,  8,  8,  5, 13,  9, 11, 13, 13, 12, 13, 11, 12,  5, 11],
       [11,  9, 13, 14,  6,  7,  6, 14, 10,  6,  8, 14, 14, 14, 14],
       [10, 12,  6,  7,  8,  6, 10,  9, 13,  6, 14, 10, 12, 10, 10],
       [10, 12, 10,  9, 11, 14,  9,  6,  7, 13,  6, 11,  8, 11,  8],
       [13, 14,  7, 14,  6, 14,  6,  8, 14,  7, 14, 12,  8,  5, 10],
       [13,  5,  9,  7,  5,  9, 13, 10, 13,  7,  7,  9, 14, 13, 11]])

>>> half_width = 3
```

Solving Jack's problem


```
# These two indices ensure that we take a slice at each (i, j) position
>>> idx_i = np.arange(ni)[:, np.newaxis, np.newaxis]
>>> idx_j = np.arange(nj)[np.newaxis, :, np.newaxis]

# This is the substantive part that picks out the window
>>> idx_k = k[:, :, np.newaxis] + \
...          np.arange(-half_width, half_width+1) # (10, 15, 7)

>>> block[idx_i, idx_j, idx_k] # slice!
```

Applying the broadcasting rules:

```
(ni,  1, 1                 ) # idx_i
(1 , nj, 1                 ) # idx_j
(ni, nj, 2*half_width + 1 ) # idx_k
---------------------------
(ni, nj, 7) <-- this is what we wanted!
```

Solution verification


```
>>> slices = cube[idx_i,idx_j,idx_k]
>>> slices.shape
(10, 15, 7)

# Now verify that our window is centered on k everywhere:
>>> slices[:,:,3]
array([[ 6,  9, 11, 10,  9, 10,  8, 13, 10, 12, 13,  9, 12,  5,  6],
       [ 7,  9,  6, 14, 11,  8, 12,  7, 12,  9,  7,  9,  8, 10, 13],
       [10, 14,  9, 13, 12, 11, 13,  6, 11,  9, 14, 12,  6,  8, 12],
       [ 5, 11,  8, 14, 10, 10, 10,  9, 10,  5,  7, 11,  9, 13,  8],
       [ 7,  8,  8,  5, 13,  9, 11, 13, 13, 12, 13, 11, 12,  5, 11],
       [11,  9, 13, 14,  6,  7,  6, 14, 10,  6,  8, 14, 14, 14, 14],
       [10, 12,  6,  7,  8,  6, 10,  9, 13,  6, 14, 10, 12, 10, 10],
       [10, 12, 10,  9, 11, 14,  9,  6,  7, 13,  6, 11,  8, 11,  8],
       [13, 14,  7, 14,  6, 14,  6,  8, 14,  7, 14, 12,  8,  5, 10],
       [13,  5,  9,  7,  5,  9, 13, 10, 13,  7,  7,  9, 14, 13, 11]])

>>> (slices[:,:,3] == k).all()
True
```

The __array_interface__

Array interface overview

Any object that exposes a suitable dictionary named `__array_interface__` may be converted to a NumPy array. This is very handy for exchanging data with external libraries. The array interface has the following important keys (see `http://docs.scipy.org/doc/numpy/reference/arrays.interface`

- **shape**
- **typestr**: see above URL for valid typecodes
- **data**: (20495857, True); 2-tuple—pointer to data and boolean to indicate whether memory is read-only
- **strides**
- **version**: 3

Discussion, questions & exercises