Setup

- Tutorial layout
- Setup

The NumPy ndarray
- Broadcasting
- Indexing
- Structured arrays
- Universal functions
- The __array_interface__
- Optimisation
- Update, wrap-up & questions

```python
import numpy as np
def print np.__version__  # version 1.3 or greater

Point your browser to the problem set at
http://mentat.za.net/numpy/kittens
```
The NumPy ndarray
Revision: Structure of an ndarray

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

```c
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 */
    PyObject *weakreflist;    /* For weakreferences */
    PyArray_Descr *descr;     /* Pointer to type struct */
    int flags;                /* Flags */
} PyArrayObject;
```
A homogeneous container

```c
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'
```c
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

```python
defining 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])

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

```c
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.

```c
PyObject* PyObject_FromVoidPtr(void* obj, void (*destr)(void *))

*Return value: New reference.*
Create a `PyObject` from the `void * obj`. The `destr` function will be called when the object is reclaimed, unless it is `NULL`.
```

Problem Set P1
Broadcasting

• Tutorial layout
• Setup

The NumPy ndarray

Broadcasting
• Broadcasting overview (1D)
• Broadcasting overview (2D)
• Broadcasting overview (3D)
• Broadcasting Rules
• Explicit broadcasting

Indexing

Structured arrays

Universal functions

The __array_interface__

Optimisation

Update, wrap-up & questions
Combining of differently shaped arrays without creating large intermediate arrays:

```python
>>> x = np.arange(4)
>>> x = np.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)
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:

<table>
<thead>
<tr>
<th></th>
<th>Scalar</th>
<th>2D</th>
<th>3D</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>(3, 4)</td>
<td>(3, 5, 1)</td>
<td>(3, 5, 2)</td>
<td></td>
</tr>
<tr>
<td>(3, )</td>
<td>(3, 1)</td>
<td>(8)</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>(3, )</td>
<td>(3, 4)</td>
<td>(3, 5, 8)</td>
<td>XXX</td>
<td></td>
</tr>
</tbody>
</table>
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]]),
     array([[1, 1, 1],
            [2, 2, 2],
            [3, 3, 3]])]
Problem Set P2
Indexing

• Tutorial layout
• Setup

The NumPy ndarray

Broadcasting

Indexing
• Jack’s Dilemma
• Jack’s Dilemma (cont’d)
○ Output shape of an indexing op
○ Output shape of an indexing op (cont’d)
• Test setup for Jack’s problem
• Solving Jack’s problem
• Solution verification

Structured arrays

Universal functions

The __array_interface__

Optimisation

Update, wrap-up & questions
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)

- Tutorial layout
- Setup

The NumPy ndarray

Broadcasting

Indexing
- Jack’s Dilemma
  - Jack’s Dilemma (cont’d)

Output shape of an indexing op
  - Output shape of an indexing op (cont’d)

Test setup for Jack’s problem
- Solving Jack’s problem
- Solution verification

Structured arrays

Universal functions

The __array_interface__

Optimisation

Update, wrap-up & questions
Remember that ndarray can be indexed in two ways:

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

Simple fancy indexing example:

```python
>>> 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.

```python
>>> 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:

```python
x[5:10, index_one, :, index_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

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

(3, 2, 5, 16)
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,  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
Problem Set P3
Structured arrays
Repeating what we said 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

<table>
<thead>
<tr>
<th>Time</th>
<th>Size</th>
<th>Position</th>
<th>Type</th>
<th>ID</th>
<th>Gain</th>
<th>Samples (2048)</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1172581077060</td>
<td>4108</td>
<td>0.715594</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>561</td>
</tr>
<tr>
<td>1172581077091</td>
<td>4108</td>
<td>0.706876</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>7</td>
</tr>
<tr>
<td>1172581077123</td>
<td>4108</td>
<td>0.698157</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>49</td>
</tr>
<tr>
<td>1172581077153</td>
<td>4108</td>
<td>0.689423</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-55</td>
</tr>
<tr>
<td>1172581077184</td>
<td>4108</td>
<td>0.680683</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-719</td>
</tr>
<tr>
<td>1172581077215</td>
<td>4108</td>
<td>0.671956</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-1503</td>
</tr>
<tr>
<td>1172581077245</td>
<td>4108</td>
<td>0.663232</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-2731</td>
</tr>
<tr>
<td>1172581077276</td>
<td>4108</td>
<td>0.654511</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-3493</td>
</tr>
<tr>
<td>1172581077306</td>
<td>4108</td>
<td>0.645787</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-3255</td>
</tr>
<tr>
<td>1172581077339</td>
<td>4108</td>
<td>0.637058</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-2303</td>
</tr>
<tr>
<td>1172581077370</td>
<td>4108</td>
<td>0.628321</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-1495</td>
</tr>
<tr>
<td>1172581077402</td>
<td>4108</td>
<td>0.619599</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-955</td>
</tr>
<tr>
<td>1172581077432</td>
<td>4108</td>
<td>0.61087</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-875</td>
</tr>
<tr>
<td>1172581077463</td>
<td>4108</td>
<td>0.602148</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>-491</td>
</tr>
<tr>
<td>1172581077497</td>
<td>4108</td>
<td>0.593438</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>167</td>
</tr>
<tr>
<td>1172581077547</td>
<td>4108</td>
<td>0.584696</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>1007</td>
</tr>
<tr>
<td>1172581077599</td>
<td>4108</td>
<td>0.575972</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>1261</td>
</tr>
<tr>
<td>1172581077650</td>
<td>4108</td>
<td>0.567244</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>1537</td>
</tr>
<tr>
<td>1172581077700</td>
<td>4108</td>
<td>0.558511</td>
<td>-0.148407</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>1105</td>
</tr>
</tbody>
</table>
Reading data from file

Reading this kind of data can be somewhat troublesome:

```matlab
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;
    end
    % 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:

```python
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']
```
Problem Set P4
Universal functions

The NumPy ndarray
Broadcasting
Indexing
Structured arrays

Universal functions
• Build your own ufuncs

The __array_interface__
Optimisation

Update, wrap-up & questions
Build your own ufuncs

- Demo ufuncs using Cython. Participants who have Cython installed may implement their own ufunc.
Problem Set P5
The __array_interface__

• Array interface overview

Optimisation

Update, wrap-up & questions
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 other libraries (e.g., PIL ↔ SciPy). The array interface has the following important keys (see https://docs.scipy.org/doc/numpy/reference/arrays.interface.html):

- **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
Problem Set P6
Optimisation

- Tutorial layout
- Setup
- The NumPy ndarray
- Broadcasting
- Indexing
- Structured arrays
- Universal functions
- The __array_interface__
- Optimisation
d- Optimisation demos

Update, wrap-up & questions
Optimisation demos

- Tutorial layout
- Setup

The NumPy ndarray
Broadcasting
Indexing
Structured arrays
Universal functions
The __array_interface__

Optimisation
- Optimisation demos

Update, wrap-up & questions

- Talk about for-loop performance, memory use of broadcasting
- Demo Cython + numpy
- Demo profiling (line_profiler, RunSnakeRun, valgrind + kcachegrind)
Update, wrap-up & questions

- Tutorial layout
- Setup
- The NumPy ndarray
- Broadcasting
- Indexing
- Structured arrays
- Universal functions
- The __array_interface__
- Optimisation

Update, wrap-up & questions
- Development Update
Development Update

- Tutorial layout
- Setup
- The NumPy ndarray
- Broadcasting
- Indexing
- Structured arrays
- Universal functions
- The __array_interface__
- Optimisation

Update, wrap-up & questions
- Development Update
Questions / comments?