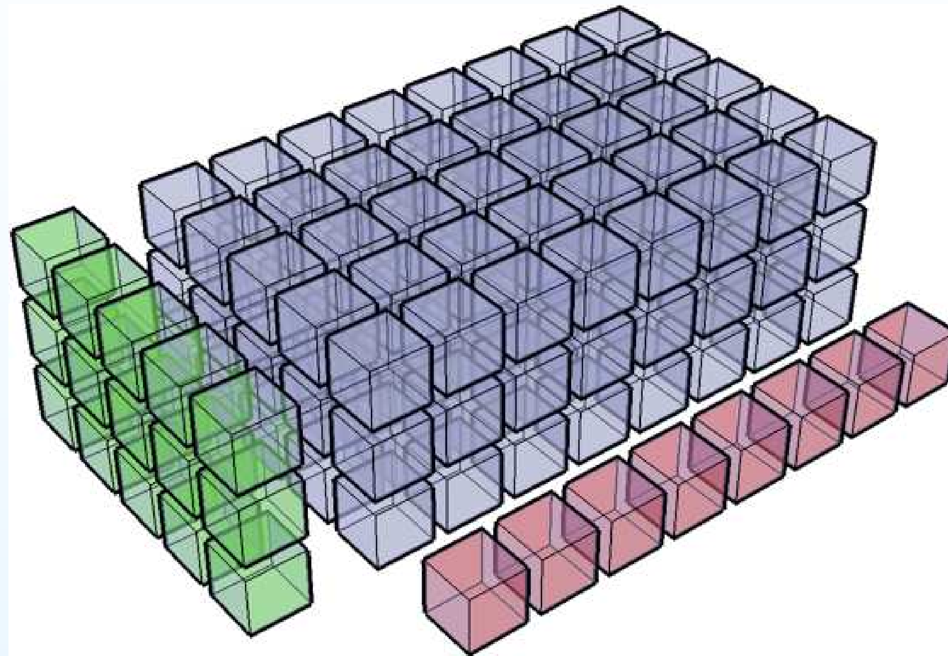
Advanced Track » NumPy



Stéfan van der Walt   David Cournapeau

18 August 2009

Overview

• Overview
Taking apart an ndarray

An ndarray is a **homogeneous container,** containing zero or more objects of a certain **data-type**. The array has a **shape** that determines its **dimensionality**. In memory, items are simply stored one after the other (i.e., a one-dimensional array). To represent an N-dimensional array, we have to know how many bytes to skip (in memory) to advance along each dimension. This concept is known as **striding** and allows the representation of non-contiguous or even repeating arrays. Functions that operate on each item in an array are known as universal functions, or simply **ufuncs**. Recently, NumPy introduced **generalised ufuncs**, which may consume more than one input value and yield more than one output value.

Before we start:

```
import numpy as np
print np.__version__ # version 1.2 or greater
```

- Overview

Taking apart an ndarray

- Anatomy
- Data buffers
- Dimensions
- Data-type
- Reading/writing data
- Strides
- Flags
- Base Pointer
- Generalised Universal Functions
- Handover

Taking apart an ndarray

- Overview

Taking apart an ndarray
- Anatomy
- Data buffers
- Dimensions
- Data-type
- Reading/writing data
- Strides
- Flags
- Base Pointer
- Generalised Universal Functions
- Handover

Anatomy of an ndarray

Straight from the bowels of `ndarrayobject.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 */
  PyObject *base;             /* Ptr to original array */
                              /* Decref this object */
                              /* on 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'
```

- Overview

Taking apart an ndarray

- Anatomy
- Data buffers
- Dimensions
- Data-type
- Reading/writing data
- Strides
- Flags
- Base Pointer
- Generalised Universal Functions
- Handover

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 */
```

Each item in the 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)
```

Data type descriptors (continued)

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	42

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']
```

«Hands on» Reading data from file

```
from StringIO import StringIO

# Setup some input data
txtdata = StringIO("""
# name   x        y        block - 2x3 ints
aaaa     1.0      8.0      1 2 3 4 5 6
aaaa     2.0      7.4      2 11 22 3 4 5 6
bbbb     3.5      8.5      3 0 22 44 5 6
aaaa     6.4      4.0      4 1 3 33 54 65
aaaa     8.8      4.1      5 5 3 4 44 77
bbbb     5.5      9.1      6 3 4 5 0 55
bbbb     7.7      8.5      7 2 3 4 5 66
""")

# Construct the data-type
# In IPython type np.dtype?<ENTER> for examples, e.g.,
#
# np.dtype([('x', np.float), ('y', np.int), ('z', np.uint8)])

dt = np.dtype(...) # Modify this line to give the correct answer

# Load data with loadtxt
data = np.loadtxt(txtdata, dtype=dt)

print data
```

«Solution» Reading data from file

```
from StringIO import StringIO

txtdata = StringIO("""
# name   x       y       block - 2x3 ints
aaaa     1.0     8.0     1 2 3 4 5 6
aaaa     2.0     7.4     2 11 22 3 4 5 6
bbbb     3.5     8.5     3 0 22 44 5 6
aaaa     6.4     4.0     4 1 3 33 54 65
aaaa     8.8     4.1     5 5 3 4 44 77
bbbb     5.5     9.1     6 3 4 5 0 55
bbbb     7.7     8.5     7 2 3 4 5 66
""")

dt = np.dtype([('name', 'S4'), ('x', float), ('y', float),
               ('block', (int, 6))])
#dt = np.dtype([('name', 'S4'), ('x', float), ('y', float),
#               ('block', (int, (2, 3)))])

data = np.loadtxt(txtdata, dtype=dt)
```

- Overview

Taking apart an ndarray

- Anatomy
- Data buffers
- Dimensions
- Data-type
- Reading/writing data
- Strides
- Flags
- Base Pointer
- Generalised Universal Functions
- Handover

Writing data to file

Load and save NumPy arrays using **np.load** and **np.save** / **np.savez:**

```
In [18]: x = np.arange(12).reshape((4,3))

In [19]: y = np.array([5, 9, 10])

In [20]: np.savez('/tmp/myarrs', x=x, y=y)

In [21]: data = np.load('/tmp/myarrs.npz')

In [22]: data['x']
Out[22]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [23]: data['y']
Out[23]: array([ 5,  9, 10])
```

- Load binary files rapidly using **np.fromfile**, as illustrated on the previous slide.

Strides

- Overview

Taking apart an ndarray

- Anatomy
- Data buffers
- Dimensions
- Data-type
- Reading/writing data
- Strides
- Flags
- Base Pointer
- Generalised Universal Functions
- Handover


```
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)
```

«Hands On» Manipulating Strides

```
# This is the function we use to view an array with new strides
as_strided = np.lib.stride_tricks.as_strided

# This is our input array:
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])

x = np.arange(12).reshape((3, 4))

# Now, provide the strides necessary to view a sliding
# 2x2 window over this array.  The output should be
#
# ([[[[ 0,  1],
#     [ 4,  5]],
#
#    [[ 1,  2],
#     [ 5,  6]],
#
#    ...

# Complete this line:
z = as_strided(x, shape=(2, 3, 2, 2),
                  strides=(..., ..., ..., ...))
```

«Solution» Manipulating Strides

```
In [54]: as_strided = np.lib.stride_tricks.as_strided
In [53]: x = np.arange(12).reshape((3, 4))
In [55]: z = as_strided(x, shape=(2,  3, 2,  2),
                          strides=(16, 4, 16, 4))

# strides = (window_move_row_down, window_move_column_right,
#            inside_window_row_down, inside_window_column_right)

In [56]: z
Out[56]:
array([[[[ 0,  1],
         [ 4,  5]],

        [[ 1,  2],
         [ 5,  6]],

        [[ 2,  3],
         [ 6,  7]]],


       [[[ 4,  5],
         [ 8,  9]],

        [[ 5,  6],
         [ 9, 10]],
```



```
        [[ 6,  7],
```

Manipulating Strides: Application

```
In [8]: s = np.apply_over_axes(np.sum, z, [2, 3])

In [9]: s.squeeze()
Out[9]:
array([[10, 14, 18],
       [26, 30, 34]])
```

Flags

```
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
```

- Overview

Taking apart an ndarray

- Anatomy
- Data buffers
- Dimensions
- Data-type
- Reading/writing data
- Strides
- Flags
- Base Pointer
- Generalised Universal Functions
- Handover

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**. Finally, we trick numpy into decreasing the reference count of the special object upon deletion by assigning it to the `*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=62`.

Generalised Universal Functions

We won't have time for this during the tutorial, but please have a look at `http://projects.scipy.org/numpy/ticket/887`, or talk to me during the conference for further detail.

- Universal functions (ufuncs) consume scalars and produce scalars (e.g. $\cos(x)$).
- Generalised ufuncs may consume more than one value and produce more than one value.
- These ufuncs can be implemented in Cython – no C hacking required! See `http://wiki.cython.org/MarkLodato/CreatingUfuncs`.

Handover

- You now know, at a very fundamental level, how an ndarray object is constructed.
- We've seen how to create structured arrays based on more complex data-types.
- The (fairly new) numpy data storage format was used to read from and write to disk.

- Manipulating array strides allows us to manipulate large chunks of data without significant memory use. Using `np.lib.stride_tricks`, we built a sliding window for operating over a 2-dimensional array.

Please join us in further discussions on the NumPy mailing list (linked from `http://www.scipy.org`).

The next half of this tutorial will be presented by **David Cournapeau**, who will tell us a bit more about his newly implemented **neighborhood iterator**.

» » » » » »