Introduction to NumPy

Travis E. Oliphant
Electrical Engineering
Brigham Young University
Provo, Utah

http://numeric.scipy.org
NumPy

- An N-dimensional “homogeneous” array object
- Universal element-by-element function objects (ufuncs)
- Basic linear algebra
- Random number generation
- Fast Fourier transforms
- Masked arrays
- Fortran (and simple C) wrapping tool (f2py)

It builds on the original Numeric code base but adds the features developed by Numarray (plus additional features)
NumPy

- Numeric
  - started in 1995 by Jim Hugunin
  - Large user-base (33972 downloads of 24.2)
  - Data-types limited and difficult to add new data-types

- Numarray (module numarray)
  - started in 2001 by Todd Miller, Rick White, and Perry Greenfield (STSCI) --- (39115 downloads of 1.5.1)
  - Added record arrays and character arrays
  - Allowed use of index arrays to select elements of an array
  - Support for memory-mapped files
  - Slow for small arrays (Python implementation)

- NumPy (module numpy)
  - Built on the Numeric code-base
  - Enhanced data-types (14674 downloads of 0.9.8)
  - Hybrid of Numarray and Numeric
NumPy is the future

- Numeric is no longer maintained.
- Numarray developers at STSCI have stated that they will only support it for a transition period less than one year.
- NumPy includes compatibility layers for both Numeric and Numarray.
- The community behind NumPy is vibrant and growing.
- Guide to NumPy a book which covers the system rather completely is available for purchase now but will be completely free in at most 4 years.
http://www.trelgol.com
What is NumPy?

- Python is a fabulous language
  - Easy to extend
  - Great syntax which encourages easy to write and maintain code
  - Incredibly large standard-library and third-party tools

- **No built-in multi-dimensional array** (but it supports the needed syntax for extracting elements from one)

- NumPy provides a **fast** built-in object (ndarray) which is a multi-dimensional array of a homogeneous data-type.
A NumPy array is a homogeneous collection of “items” of the same “data-type” (dtype)

```python
>>> import numpy as N
>>> a = N.array([[1,2,3],[4,5,6]],float)
>>> print a
[[1. 2. 3.]
 [4. 5. 6.]]
>>> print a.shape, "\n", a.itemsize
(2, 3) 8
>>> print a.dtype, a.dtype.type
'<f8' <type 'float64scalar'>
>>> type(a[0,0])
<type 'float64scalar'>
>>> type(a[0,0]) is type(a[1,2])
True
```
Every dimension of a ndarray is accessed by stepping (striding) a fixed number of bytes through memory.

If memory is contiguous, then the strides are “pre-computed” indexing-formulas for either Fortran-order (first-dimension varies the fastest), or C-order (last-dimension varies the fastest) arrays.
Array slicing (Views)

- Memory model allows “simple indexing” (integers and slices) into the array to be a view of the same data.

```python
>>> b = a[:,::2]
>>> b[0,1] = 100
>>> print a
[[   1.    2.  100.]
 [   4.    5.    6.]]
>>> c = a[:,::2].copy()
>>> c[1,0] = 500
>>> print a
[[   1.    2.  100.]
 [   4.    5.    6.]]
```

Other uses of view

```python
>>> b = a.view('i8')
>>> [hex(val.item()) for val in b.flat]
['0x3FF0000000000000L',
 '0x4000000000000000L',
 '0x4059000000000000L',
 '0x4010000000000000L',
 '0x4014000000000000L',
 '0x4018000000000000L']
```
There are two related concepts of “type”
- The data-type object (dtype)
- The Python “type” of the object created from a single array item (hierarchy of scalar types)

The **dtype** object provides the details of how to interpret the memory for an item. It's an instance of a single dtype class.

The “type” of the extracted elements are true Python classes that exist in a hierarchy of Python classes (similar to Numarray).

Every dtype object has a type attribute which provides the Python object returned when an element is selected from the array.
Built-in "scalar" types

- bool
- object
- number
  - integer
    - signedinteger
      - byte
      - short
      - int
      - longlong
    - unsignedinteger
      - ubyte
      - ushort
      - uint
      - ulonglong
  - inexact
    - floating
    - complexfloating
    - single
    - float
    - longfloat
    - csingle
    - complex
    - clongfloat
  - character
    - str
    - unicode
  - void
Data-type object (dtype)

- There are 21 “built-in” (static) data-type objects
- New (dynamic) data-type objects are created to handle
  - Alteration of the byteorder
  - Change in the element size (for string, unicode, and void built-ins)
  - Addition of fields
  - Change of the type object (only allowed for sub-classes of the void-scalar)
- Creation of data-types is quite flexible.
- New user-defined “built-in” data-types can also be added (but must be done in C and involves filling a function-pointer table)
Data-type fields

- An item can include fields of different data-types.
- A field is described by a data-type object and a byte offset --- this definition allows nested records.
- The array construction command interprets tuple elements as field entries.

```python
>>> dt = N.dtype("i4,f8,a5")
>>> print dt.fields
{'f1': (dtype('<i4'), 0), 'f2': (dtype('<f8'), 4), 'f3': (dtype('|S5'), 12))
>>> a = N.array([(1,2.0,"Hello"), (2,3.0,"World")], dtype=dt)
>>> print a['f3']
[Hello World]
```
## Array attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Buffer object representing memory</td>
</tr>
<tr>
<td>dtype</td>
<td>Data-type object</td>
</tr>
<tr>
<td>flags</td>
<td>Flags object (e.g. contiguous, aligned, writeable)</td>
</tr>
<tr>
<td>flat</td>
<td>1D iterator object</td>
</tr>
<tr>
<td>imag</td>
<td>Imaginary part or read-only zeros</td>
</tr>
<tr>
<td>real</td>
<td>Real part</td>
</tr>
<tr>
<td>T</td>
<td>Transpose view</td>
</tr>
<tr>
<td>base</td>
<td>Memory-exposing object</td>
</tr>
<tr>
<td>ctypes</td>
<td>Object for ctypes interfacing</td>
</tr>
<tr>
<td>itemsize</td>
<td>Bytes in each item</td>
</tr>
<tr>
<td>size</td>
<td>Number of items</td>
</tr>
<tr>
<td>nbytes</td>
<td>Number of bytes</td>
</tr>
<tr>
<td>ndim</td>
<td>Number of dimensions</td>
</tr>
<tr>
<td>shape</td>
<td>Tuple showing shape</td>
</tr>
<tr>
<td>strides</td>
<td>Tuple showing strides</td>
</tr>
</tbody>
</table>

**bold**: can be set
## Array Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>astype</code></td>
<td><code>(dtype &lt;None&gt;)</code></td>
<td>Cast to another data type</td>
</tr>
<tr>
<td><code>byteswap</code></td>
<td><code>(inplace &lt;False&gt;)</code></td>
<td>Byteswap array elements</td>
</tr>
<tr>
<td><code>copy</code></td>
<td>()</td>
<td>Copy array</td>
</tr>
<tr>
<td><code>dump</code></td>
<td><code>(file)</code></td>
<td>Pickle to stream or file</td>
</tr>
<tr>
<td><code>dumps</code></td>
<td>()</td>
<td>Get pickled string</td>
</tr>
<tr>
<td><code>fill</code></td>
<td><code>(scalar)</code></td>
<td>Fill an array with scalar value</td>
</tr>
<tr>
<td><code>getfield</code></td>
<td><code>(dtype=, offset=0)</code></td>
<td>Return a field of the array</td>
</tr>
<tr>
<td><code>setflags</code></td>
<td><code>(write=None, align=None, uic=None)</code></td>
<td>Set array flags</td>
</tr>
<tr>
<td><code>tofile</code></td>
<td><code>(file=, sep=' ', format='')</code></td>
<td>Raw write to file</td>
</tr>
<tr>
<td><code>tolist</code></td>
<td>()</td>
<td>Array as a nested list</td>
</tr>
<tr>
<td><code>item</code></td>
<td>()</td>
<td>Python scalar from first element</td>
</tr>
<tr>
<td><code>tostring</code></td>
<td><code>(order='C')</code></td>
<td>String of raw memory</td>
</tr>
<tr>
<td><code>view</code></td>
<td><code>(obj)</code></td>
<td>View as another data type or class</td>
</tr>
</tbody>
</table>
## Array methods

### Item selection and shape manipulation

<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argsort</td>
<td>(axis=None, kind='quick')</td>
<td>Indices showing how to sort array.</td>
</tr>
<tr>
<td>choose</td>
<td>(c0, c1, ..., cn, out=None, clip='raise')</td>
<td>Choose from different arrays based on value of :</td>
</tr>
<tr>
<td>compress</td>
<td>(condition=, axis=None, out=None)</td>
<td>Elements of self where condition is true.</td>
</tr>
<tr>
<td>diagonal</td>
<td>(offset=0, axis1=0, axis2=1)</td>
<td>Return a diagonal from self.</td>
</tr>
<tr>
<td>flatten</td>
<td>()</td>
<td>A 1-d copy of self.</td>
</tr>
<tr>
<td>nonzero</td>
<td>()</td>
<td>True where self is not zero.</td>
</tr>
<tr>
<td>put</td>
<td>(indices=, values=, mode='raise')</td>
<td>Place values at 1-d index locations of self.</td>
</tr>
<tr>
<td>putmask</td>
<td>(mask=, values=)</td>
<td>Place values in 1-d index locations where mask</td>
</tr>
<tr>
<td>ravel</td>
<td>()</td>
<td>1-d version of self (no data copy if self is C-style</td>
</tr>
<tr>
<td>repeat</td>
<td>(repeats=, axis=None)</td>
<td>Repeat elements of self.</td>
</tr>
<tr>
<td>reshape</td>
<td>(d1, d2, ..., dn, order='C')</td>
<td>Return reshaped version of self.</td>
</tr>
<tr>
<td>resize</td>
<td>(d1, d2, ..., dn, refcheck=1, order='Any')</td>
<td>Resize self in-place.</td>
</tr>
<tr>
<td>searchsorted</td>
<td>(values)</td>
<td>Show where values would be placed in self (ass</td>
</tr>
<tr>
<td>sort</td>
<td>(axis=None, kind='quick')</td>
<td>Copy of self sorted along axis.</td>
</tr>
<tr>
<td>squeeze</td>
<td>()</td>
<td>Squeeze out all length-1 dimensions.</td>
</tr>
<tr>
<td>swapaxes</td>
<td>(axis1, axis2)</td>
<td>Swap two dimensions of self.</td>
</tr>
<tr>
<td>take</td>
<td>(indices=, axis=None, out=None, mode='raise')</td>
<td>Select elements of self along axis according to :</td>
</tr>
</tbody>
</table>
## Array methods

### Array Calculation

<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>(axis=None)</td>
<td>true if all entries are true.</td>
</tr>
<tr>
<td>any</td>
<td>(axis=None)</td>
<td>true if any entries are true.</td>
</tr>
<tr>
<td>argmax</td>
<td>(axis=None)</td>
<td>index of largest value.</td>
</tr>
<tr>
<td>argmin</td>
<td>(axis=None)</td>
<td>index of smallest value.</td>
</tr>
<tr>
<td>clip</td>
<td>(min=, max=)</td>
<td>self[self&gt;max]=max; self[self&lt;min]=min</td>
</tr>
<tr>
<td>conj</td>
<td>()</td>
<td>complex conjugate</td>
</tr>
<tr>
<td>cumprod</td>
<td>(axis=None, dtype=None)</td>
<td>cumulative product</td>
</tr>
<tr>
<td>cumsum</td>
<td>(axis=None, dtype=None)</td>
<td>cumulative sum</td>
</tr>
<tr>
<td>max</td>
<td>(axis=None)</td>
<td>maximum of self</td>
</tr>
<tr>
<td>mean</td>
<td>(axis=None, dtype=None)</td>
<td>mean of self</td>
</tr>
<tr>
<td>min</td>
<td>(axis=None)</td>
<td>minimum of self</td>
</tr>
<tr>
<td>prod</td>
<td>(axis=None, dtype=None)</td>
<td>multiply elements of self together</td>
</tr>
<tr>
<td>ptp</td>
<td>(axis=None, dtype=None)</td>
<td>self.max(axis)-self.min(axis)</td>
</tr>
<tr>
<td>var</td>
<td>(axis=None, dtype=None)</td>
<td>variance of self</td>
</tr>
<tr>
<td>std</td>
<td>(axis=None, dtype=None)</td>
<td>standard deviation of self</td>
</tr>
<tr>
<td>sum</td>
<td>(axis=None, dtype=None)</td>
<td>add elements of self together</td>
</tr>
<tr>
<td>trace</td>
<td>(offset, axis1=0, axis2=0,</td>
<td>Sum along a diagonal</td>
</tr>
<tr>
<td></td>
<td>dtype=None)</td>
<td></td>
</tr>
</tbody>
</table>
Universal Functions

- ufuncs are objects that rapidly evaluate a function element-by-element over an array.
- Core piece is a 1-d loop written in C that performs the operation over the largest dimension of the array.
- For 1-d arrays it is equivalent to but much faster than list comprehension.

```python
>>> type(N.exp)
<type 'numpy.ufunc'>
>>> x = array([1,2,3,4,5])
>>> print N.exp(x)
[  2.71828183   7.3890561    20.08553692   54.59815003  148.4131591 ]
>>> print [math.exp(val) for val in x]
[2.7182818284590451, 7.3890560989306504, 20.085536923187668, 54.598150033144236, 148.4131591025766]
```
Broadcasting

- When there are multiple inputs, then they all must be “broadcastable” to the same shape.
  - All arrays are promoted to the same number of dimensions (by pre-pending 1's to the shape)
  - All dimensions of length 1 are expanded as determined by other inputs with non-unit lengths in that dimension.

```
>>> x = [1,2,3,4];
>>> y = [[10],[20],[30]]
>>> print N.add(x,y)
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
>>> x = array(x)
>>> y = array(y)
>>> print x+y
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

x has shape (4,) the ufunc sees it as having shape (1,4)

y has shape (3,1)

The ufunc result has shape (3,4)
Available ufuncs

absolute  add  arccos  arccosh  arcsin  arcsinh  arctan  arctanh
bitwise_and  bitwise_or  bitwise_xor  ceil  conj  conjugate  cos  cosh  divide
equal  exp  expm1  fabs  floor  floor_divide  fmod  frexp  greater  greater_equal  hypot  invert  isfinite  isnan  ldexp  left_shift  less
less_equal  log  log10  log1p  logical_and  logical_not  logical_or  logical_xor  maximum  minimum  mod  modf  multiply  negative  not_equal  ones_like  power  reciprocal
remainder  right_shift  rint  sign  signbit  sin  sinh  sqrt  square  subtract  tan  tanh  true_divide
Array Interface

- How do different Python modules share array information?
  - Put NumPy in the Python standard library?
  - Require installation of NumPy?
  - **Use the array interface**

http://numeric.scipy.org/array_interface.html
Demo

- Array creation
- Array math
- FFT
- Eigen-decomposition
- Random-number generation
- f2py
Other Tools

- http://www.scipy.org

Python

SciPy

IPython

Matplotlib

NumPy

PIL

VTK

Tkinter

ETS

MolKit

PMV

MayaVi

DejaVu

PMV

ViPER

Pygame