NUMPY BASICS

Numpy

- NumPy is the fundamental package for scientific computing with Python.
- It contains:
  - a powerful N-dimensional array object
  - tools for integrating C/C++ and Fortran code
- Uses pre-compiled C codes, so you can get C-like speed

Numpy

- Numpy provides fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

ndarray

- At the core of the NumPy package, is the *ndarray* object.
- This encapsulates *n*-dimensional arrays of homogeneous data types

Numpy ndarray vs. Python Sequences

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically).
- Changing the size of an *ndarray* will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.
  - The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

Less Code, Less Errors

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

```
for (i = 0; i < rows; i++): {
  c[i] = a[i]*b[i];
}
```

```
for (i = 0; i < rows; i++): {
  for (j = 0; j < columns; j++): {
    c[i][j] = a[i][j]*b[i][j];
  }
}
```

```
c = a * b
```

- Best of both worlds
  - Coding flexibility of Python
  - Near C Speed

ndarray

- Also has an alias *array*
- Indexed by tuples
- Dimensions are called axes
  - The number of axes is called *rank*

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

Important Attributes

ndarray.ndim

the number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as *rank*.

ndarray.shape

the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, `shape` will be `(n,m)`. The length of the `shape` tuple is therefore the rank, or number of dimensions, `ndim`.

ndarray.size

the total number of elements of the array. This is equal to the product of the elements of `shape`.

ndarray.dtype

an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

Important Attributes

ndarray.itemsize

the size in bytes of each element of the array. For example, an array of elements of type `float64` has `itemsize` 8 (=64/8), while one of type `complex32` has `itemsize` 4 (=32/8). It is equivalent to `ndarray.dtype.itemsize`.

ndarray.data

the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

Important Attributes

```
In[4]: nd = np.array([[1,2,3],[4,5,6]])
In[5]: nd.ndim
Out[5]:
2
In[6]: nd.shape
Out[6]:
(2, 3)
In[7]: nd.size
Out[7]:
6
In[8]: nd.dtype
Out[8]:
dtype('int32')
In[9]: nd.itemsize
Out[9]:
4
In[10]: nd.data
Out[10]:
<memory at 0x0000022CEEC201F8>
```

```
In[11]: nd
Out[11]:

array([[1, 2, 3],
       [4, 5, 6]])
In[12]: type(nd)
Out[12]:
numpy.ndarray
```

```
In[13]: nd.dtype.name
Out[13]:
'int32'
```

Example

```
In[17]: nd = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
In[18]: nd
Out[18]:

array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
In[19]: nd.ndim
Out[19]:
3
In[20]: nd.shape
Out[20]:
(2, 2, 3)
In[21]: nd.size
Out[21]:
12
```

Array Creation From List

- Type is derived from data type of list element
- Can also be specified explicitely

```
In[22]: nd = np.array([1,2.3])
In[23]: nd.dtype
Out[23]:
dtype('float64')
In[24]: nd = np.array([1,2])
In[25]: nd.dtype
Out[25]:
dtype('int32')
In[26]: nd = np.array([1,2],dtype='int64')

In[28]: nd.dtype
Out[28]:
dtype('int64')
```

Array Creation From List

```
In[29]: nd = np.array([1,2],dtype=np.float16)
In[30]: nd.dtype
Out[30]:
dtype('float16')
In[31]: nd
Out[31]:
array([ 1.,  2.], dtype=float16)
```

Array creation with predefined size

- Changing array size/ Growing array is expensive
  - New array is created
- Check out these functions
  - np.zeros(), np.ones(), np.empty()
    - All take the shape as input as a tuple
  - np.zeros_like(), ~~np.ones:like()~~, ~~np.emplty_like()~~

np.ones_like() np.empty_like()

Array creation

- np.arange()
  - like range()
  - takes float as argument

- np.linspace()

- also check :
  np.random.random()
  reshape()

```
In[51]: np.arange(6)
Out[51]:
array([0, 1, 2, 3, 4, 5])
In[52]: np.arange(1,6)
Out[52]:
array([1, 2, 3, 4, 5])
In[53]: np.arange(1,6,2)
Out[53]:
array([1, 3, 5])
In[54]: np.arange(1,6,2.5)
Out[54]:
array([ 1. ,  3.5])
In[55]: np.arange(0,10,0.25)
```

Array Creation From Function

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
... )
```

Array Printing

- The last axis is print from left to right
- The rest are printed top to bottom

Basic Operations

- Most operations work element wise
- + - * / **
- += -= *=
- a < 4 a > 10
  - Return Boolean

Upcasting

When operating with arrays of different types, the type of the resulting array corresponds to the more general

Basic Operations

- Some other operations are provided as a member method within the ndarray class
  - a.sum()
  - a.sum(axis=0)
  - a.min()
  - a.max()
  - a.mean()
  - a.dot() #matrix Multiplication

Universal Functions

- Some basic methods are provided as functions in np: universal functions
  - np.sin() np.cos() np.tan()
  - np.exp()
  - np.arange()
  - np.sqrt()

Indexing, Slicing and Iterating

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
```

```
>>> b[0:5, 1]                       # each row in the second column of b
array([ 1, 11, 21, 31, 41])
```

```
>>> b[ : ,1]                        # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                      # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Indexing, Slicing and Iterating

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
>>> b[-1]                                   # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

The **dots** (...) represent as many colons as needed to produce a complete indexing tuple. For example, if x is a rank 5 array (i.e., it has 5 axes), then

- x[1,2,...] is equivalent to x[1,2,:,:,:],
- x[...,3] to x[:,:,:,:,3] and
- x[4,...,5,:] to x[4,:,:,5,:].

```
>>> c = np.array( [[[  0,  1,  2],               # a 3D array (two stacked 2D arrays)
...                 [ 10, 12, 13]],
...                [[100,101,102],
...                 [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                   # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                   # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

Iteration

- starts from leftmost(first) axis

```
In[165]: a
Out[165]:

array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
In[166]: for i in a:
    ...:     print(i)
    ...:
[0 1 2 3 4]
[5 6 7 8 9]
[10 11 12 13 14]
```

```
In[162]: a
Out[162]:

array([[[ 0.,  1.,  2.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 2.,  3.,  4.,  5.]],

       [[ 0.,  1.,  2.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 2.,  3.,  4.,  5.]]])
In[163]: for i in a:
    ...:     print(i)
    ...:
[[ 0.  1.  2.  3.]
 [ 1.  2.  3.  4.]
 [ 2.  3.  4.  5.]]
[[ 0.  1.  2.  3.]
 [ 1.  2.  3.  4.]
 [ 2.  3.  4.  5.]]
```

Iteration a.flat and a.flatten()

```
In[169]: a
Out[169]:

array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
In[170]: for i in a.flat:
    ...:     print(i,end=' ')
    ...:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 In[171]: a.flatten()
Out[171]:
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

ravel() and flatten()

```
import numpy as np
y = np.array(((1,2,3),(4,5,6),(7,8,9)))
OUTPUT:
print(y.flatten())
[1   2   3   4   5   6   7   8   9]
print(y.ravel())
[1   2   3   4   5   6   7   8   9]
```

The current API is that:

- `flatten` always returns a copy.
- `ravel` returns a view of the original array whenever possible. This isn't visible in the printed output, but if you modify the array returned by ravel, it may modify the entries in the original array. If you modify the entries in an array returned from flatten this will never happen. ravel will often be faster since no memory is copied, but you have to be more careful about modifying the array it returns.

resize() and reshape()

- reshape() returns new array with modified size
- resize() changes the array

reshape()

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
In[195]: a.reshape(5,3)
Out[195]:

array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

- Transpose
  - a.T

np.vstack() np.hstack()

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

For arrays of with more than two dimensions, hstack stacks along their second axes, vstack stacks along their first axes

Inserting into array

```
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([1, 5, 1, 2, 2, 3, 3])
>>> np.insert(a, 1, 5, axis=1)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
```

Inserting into array

```
>>> np.insert(a, [1], [[1],[2],[3]], axis=1)
array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])
>>> np.array_equal(np.insert(a, 1, [1, 2, 3], axis=1),
...                np.insert(a, [1], [[1],[2],[3]], axis=1))
True
```

More in documentation:

https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.insert.html

Resources

- Numpy Documentation

Any Questions??