BRIEF INTRODUCTION TO NUMPY

CS 5010

Quick Overview

- Very useful library used for **manipulating n-dimensional arrays**
- Problem with the native Python list data structure is that it doesn't scale well for more than one dimension
- NumPy is built as an extension of the native Python **list** functionality
- Must remember to import numpy and all modules within:
  - `import numpy` [need to qualify]
    - (often: import numpy as np)
  - `from numpy import *` [*no need to qualify*]

Arrays

- An array can hold <u>many types of data</u>, not just numerical data types

```
a = array([0,1,2,3])                    # all integers
a
Out[12]: array([0, 1, 2, 3])
```

```
mix = array([0,'cat',2.5,7,'hello'])  # mixed array
mix
Out[14]:
array(['0', 'cat', '2.5', '7','hello'],
   dtype='<U11')       # Unicode character less than 11 characters
```

```
a.dtype          # checking the type of array 'a'
Out[16]: dtype('int32') # integer type
```

Arrays

arr = array([1,2,3])

charArr = array(["hi","hello","goodbye"])

arr.max()

Out[19]: 3

arr.cumsum()

Out[22]: array([1, 3, 6], dtype=int32)

arr.dtype

Out[20]: dtype('int32')

charArr

Out[21]: array(['hi', 'hello', 'goodbye'],
      dtype='<U7')

arr.size

Out[24]: 3

arr.sum()

Out[27]: 6

Single-dimension Arrays vs. Lists

- Is there any benefit of 1-D array vs list (which is 1D)
- Yes! Arrays are memory-efficient containers that provides fast numerical operations

L = range(1000)

timeit [i**2 for i in L]   # An operation on each element of L

10000 loops, best of 3: **468** µs per loop

a = **arange**(1000)   # "*a range*" not "*arrange*"

timeit a**2   # An operation on each ele of **array**

1000000 loops, best of 3: **1.76** µs per loop

NumPy Reference documentation

- Assuming: from numpy import *

- lookfor('create array')

Search results for 'create array'

----

numpy.array

    Create an array.

numpy.memmap

    Create a memory-map to an array stored in a *binary*
file on disk.

… *<more results shown>*

Manual Construction of 1- and 2-D arrays

- ndarrays (n-dimensional arrays) – equivalent to matrices in linear algebra
- **1D**:

```
a = array([1,2,3])
a               Out[4]: array([1, 2, 3])

a.ndim          Out[5]: 1
a.shape         Out[6]: (3,)
len(a)          Out[7]: 3
type(a)         Out[8]: numpy.ndarray
```

Manual Construction of 1- and 2-D arrays

- **2D**:

  b = array([(1.5, 2, 3),(4, 5, 6)])

  b

  Out[9]:

```
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

b.ndim	Out[10]: 2
b.shape	Out[11]: (2, 3)
len(b)	Out[12]: 2
type(b)	Out[13]: **numpy.ndarray**

Some Array Operations

a = array([(1.5, 2, 3),(4, 5, 6)])

a

Out[15]:

```
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

sin(a)

Out[16]:

```
array([[ 0.99749499,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ]])
```

Some Array Operations

```
a = b
b = pow(a,2)
b
Out[18]:
array([[  2.25,   4.  ,   9.  ],
       [ 16.  ,  25.  ,  36.  ]])

c = pow(a,3)
c
Out[20]:
array([[   3.375,    8.   ,   27.   ],
       [  64.   ,  125.   ,  216.   ]])
```

Data Types Examples

```
d = array([1,2,3], dtype=float)  # specify the type
d               Out[22]: array([ 1.,  2.,  3.])

type(d)         Out[23]: numpy.ndarray
d.dtype         Out[24]: dtype('float64')

e = array([True, False, False, True])
e       Out[27]: array([ True, False, False,  True], dtype=bool)
e.dtype         Out[28]: dtype('bool')
```

Data Types Examples

```
f = array(['Bonjour', 'Hello', 'Hallo',])
f
Out[30]:
array(['Bonjour', 'Hello', 'Hallo'],
      dtype=‘<U7')

f.dtype
Out[31]: dtype(<U7')     # Unicode, max. 7 letters
```

Some More Array Operations

```
j = array([[0.0, -1.0], [1.0, 0.0]])
j
Out[39]:
array([[ 0., -1.],
       [ 1.,  0.]])

dot(j,j)  # matrix product
Out[40]:
array([[-1.,  0.],
       [ 0., -1.]])
```

Some More Array Operations

- from numpy import *
- from numpy.linalg import *   # linear algebra
- a = array([[1.0, 2.0], [3.0, 4.0]])

Out[34]:

```
array([[ 1.,  2.],
       [ 3.,  4.]])
```

- **u = eye(2)**   # unit 2x2 matrix; "eye" represents "I“ (identity)

  u

  Out[37]:

```
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Some More Array Operations

a

Out[41]:

```
array([[ 1.,  2.],
       [ 3.,  4.]])
```

y = array([[5.0], [7.0]])

- **solve(a,y)** # solve a linear matrix equation. solve(a,b) where a is the coefficient matrix and b is the 'dependent variable' values. Returns solution to the system ax = b

Out[43]:

```
array([[-3.],
       [ 4.]])
```

Indexing ndarrays

- NumPy arrays can be indexed in the same way that native Python lists can
- However, NumPy arrays have additional functionalities
- The function “arange” is a useful NumPy function used to automatically create an ndarray (n-dimensional array) with numbers with certain increments (default 1xn array)

```
p = arange(12)**2
indices = array([1,2,3,7])  # return values of those indices
p[indices]              Out[65]: array([ 1,  4,  9, 49])
------------------------------------------------------------
p is array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121])
```

Indexing ndarrays

p2 = arange(10)*2 # 10 numbers, index multiply by 2

indices = array([1,2,3,7]) # return values of *those* indices

p2[indices]   Out[66]: array([2, 4, 6, 14])

----

p2 is array([0, 2, 4, 6, 8, 10, 12, 14, 16, 18])

Resizing and Modifying ndarrays

- Another useful feature of NumPy is it's ability to reshape and manipulate the dimensions of the ndarray

```
In [33]: b = arange(12)
In [34]: b
Out[34]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [35]: b.shape = 3,4    # 3 rows, 4 columns

In [36]: b
Out[36]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Resizing and Modifying ndarrays

```
In [37]: b.shape = 3,-1  #use -1, when you aren't sure
                          of dimension for either row or column

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

```
In [39]: b.shape = -1,3

In [40]: b
Out[40]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Note: the total number of datapoints must be **divisible** by the <u>specified dimension</u>, since dimensions are always integers. For example an array of **1x11** *can't* be reshaped with **3** rows or columns.

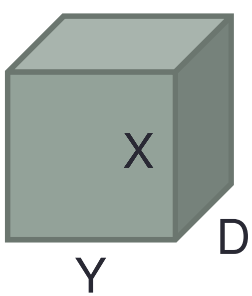
Resizing and Modifying ndarrays

- In [8]: a = arange(30)

- In [9]: a.shape = 2,-1,3 # -1 means "whatever is needed"

- In [10]: a.shape
- Out[10]: (2, 5, 3)

  D, X, Y



- In [11]: a
- Out[11]:

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

       [[15, 16, 17],
        [18, 19, 20],
        [21, 22, 23],
        [24, 25, 26],
        [27, 28, 29]]] )
```

D=depth; X=row; Y=col

Resizing and Modifying ndarrays

```
# Returning a back to the default 1D (original form):
In [20]: a = a.flatten()
In [21]: a
Out[21]:
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
```

```
In [22]: a = a.ravel()

In [23]: a
Out[23]:
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
```

```
In [24]: a.shape = 1,-1

In [25]: a
Out[25]:
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])
```

Stacking Arrays

- We can stack together different arrays. These functionalities allows different arrays to be easily "joined" or "attached" to each other, by specifying the axis
- The "**vstack**" and "**hstack**" are two important functions (stands for vertically/horizontally stack). They both accept **tuple** of the two arrays.

- p = array([[5,2],[9,6]])
  p
  Out[90]:
  ```
  array([[5, 2],
         [9, 6]])
  ```
- q = array([[1,7],[3,0]])
  q
  Out[92]:
  ```
  array([[1, 7],
         [3, 0]])
  ```

Stacking Arrays

- p,q

  Out[95]:

```
(array([[5, 2],
       [9, 6]]),
array([[1, 7],
       [3, 0]]))
```

- vstack((p,q))
- Out[96]:

```
array([[5, 2],
       [9, 6],
       [1, 7],
       [3, 0]])
```

- vstack((q,p))
- Out[97]:

```
array([[1, 7],
       [3, 0],
       [5, 2],
       [9, 6]])
```

Stacking Arrays

- hstack((p,q))

Out[98]:

```
array([[5, 2, 1, 7],
       [9, 6, 3, 0]])
```

- concatenate((p,q))
  # default axis = 0

Out[99]:

```
array([[5, 2],
       [9, 6],
       [1, 7],
       [3, 0]])
```

- concatenate((p,q), axis=1)
  # default axis = 0

Out[100]:

```
array([[5, 2, 1, 7],
       [9, 6, 3, 0]])
```

Note: there is another function called "**dstack**", which is for "depth" (this is when we have 3 axis).

For arrays with more than two dimensions, **hstack** stacks along their second axes, **vstack** stacks along their first axes, and **concatenate** (a function) allows for an optional argument giving the number of the axis along which the concatenation should happen.

Stacking Higher Dimensions

- Stacking can get complicated for arrays with dimensions greater than 3. Following is a table describing some relevant stacking functions:

`column_stack`(tup)	Stack 1-D arrays as columns into a 2-D array.
`concatenate`((a1, a2, ...)[, axis])	Join a sequence of arrays together.
`dstack`(tup)	Stack arrays in sequence depth wise (along third axis).
`hstack`(tup)	Stack arrays in sequence horizontally (column wise).
`vstack`(tup)	Stack arrays in sequence vertically (row wise).

Splitting Arrays

- Similar to stacking, there are splitting functions that split arrays anywhere based on axis and position
- `r = random.randn(4,4)` # generates a random 4x4
- `hsplit(r,4)` # split r into 4 columns
- `hsplit(r, 4)[0]` # get the first (index 0) col
- `hsplit(r, 4)[0].shape` # describe dimensions → (4,1)
- `hsplit(r,(3,4))` # Split a after the 3rd & 4th column
- `hsplit(r,(3,4))[1]` # get the second (index 1) part of split

In a similar fashion to **hsplit**, we can use **vsplit**

The Matrix Object

- A = matrix('1.0 2.0; 3.0 4.0') # Matrix: a specialized 2-D array

  A
  Out[50]:
  ```
  matrix([[ 1.,  2.],
          [ 3.,  4.]])
  ```
  To get more options: A.<tab> for more options
  A.max()
  Out[52]: 4.0

  A.T # Transpose
  Out[53]:
  ```
  matrix([[ 1.,  3.],
          [ 2.,  4.]])
  ```

Matrix Operations

- **X = matrix('5,7')**

- **B = X.T**

  B

  Out[57]:

```
matrix([[5],
        [7]])
```

- **print A*B** # matrix multiplication

```
[[ 19.]
 [ 43.]]
```

- A

  Out[59]:

```
matrix([[ 1.,  2.],
        [ 3.,  4.]])
```

- **A.std()** # standard deviation

  Out[60]: 1.1180339887498949

- **B.std()**

  Out[61]: 1.0

Can Find Additional Information At …

- SciPy.org NumPy documentation:

- http://docs.scipy.org/doc/numpy/
- http://docs.scipy.org/doc/numpy/reference/
- http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html

Next…

- A very brief introduction to Pandas!