NumPy/Scipy/Stats

Brad Cenko / Berian James
24 September 2012

Numpy for IDL or MATLAB users:

https://www.cfa.harvard.edu/~jbattat/computer/python/science/idl-numpy.html

http://www.scipy.org/NumPy_for_Matlab_Users

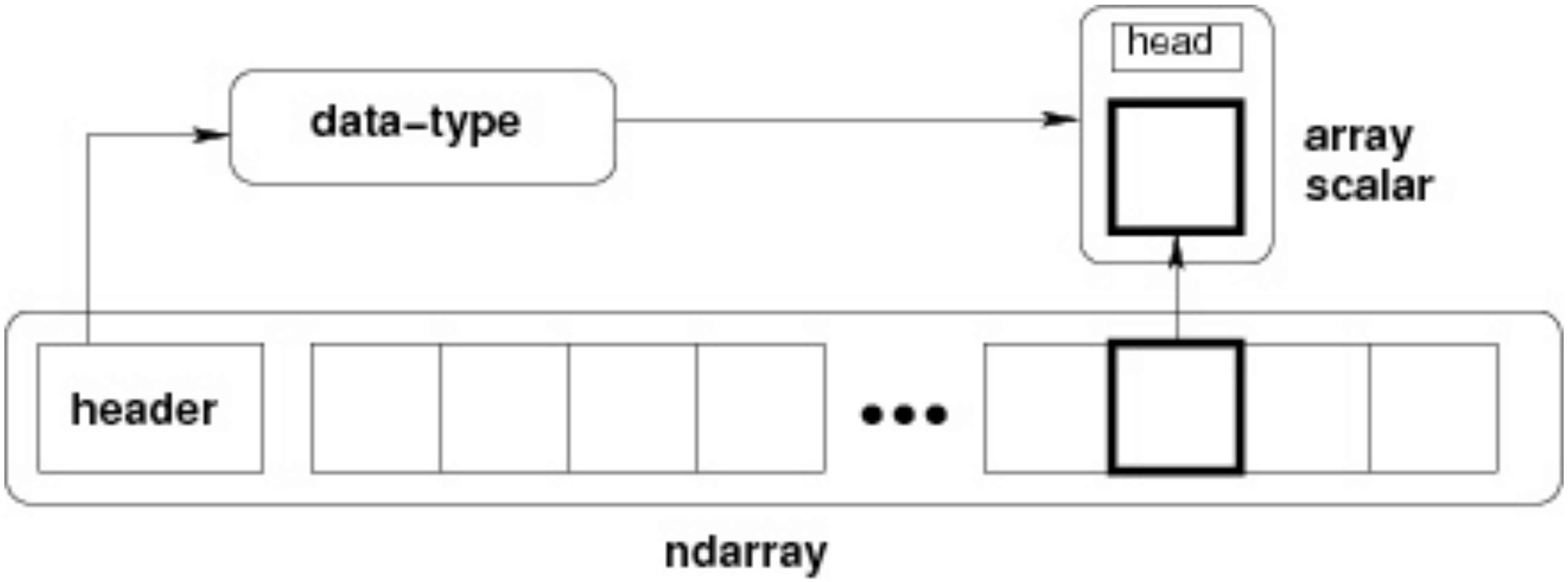
numpy methods & modules

- Array creation; Copy and referencing
- Manipulations, slicing, and indexing
- Universal functions and broadcasting
- Masked arrays

[not covering today: `distutils`, `testing, f2py]`

scipy methods & modules

ndarray class



An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

Instantiating *ndarrays*

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> b = np.ones((3,2))
>>> b
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
>>> c = np.empty((2,3,2))
>>> c
array([[[ -8.45190247e+002,   1.21278312e-062],
        [ -4.87268684e+114,  -1.45508684e-021],
        [  9.48788057e+150,   2.67670692e+015]],

       [[ -1.13067367e+192,  -3.37744342e+056],
        [  2.20318863e+228,   6.21927996e+092],
        [ -2.62364069e+269,   9.19414899e-309]]])

>>> %timeit np.zeros((10,10,13))
100000 loops, best of 3: 2.02 us per loop
>>> %timeit np.empty((10,10,13))
1000000 loops, best of 3: 1.55 us per loop
```

ndarrays are (almost) never instantiated directly, but instead using a method that returns one

Instantiating *ndarrays*

```
>>> np.linspace(1,10,5)
array([  1.   ,   3.25,   5.5 ,   7.75,  10.  ])
>>> np.logspace(1,2,5)
array([  10.,17.782,31.622,56.23413252,  100.])



>>> !cat temp.txt
1 2
3 4
>>> a = np.loadtxt("temp.txt")
>>> a
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> np.save("temp2.dat.npy",a)
>>> np.savetxt("temp3.txt", b, delimiter=",", fmt="%f")
>>> !cat temp2.dat.npy
^?NUMPYF{'descr': '<i4', 'fortran_order': False,
'shape': (2, 3), } ...
>>> !cat temp3.txt
1.000000,2.000000,3.000000,4.000000
```

linspace & logspace: *very handy*

ndarrays can also be directly read from / written to files. There are modules for csv, FITS, JPEG, wav, etc.

also, `np.savez`

scipy.io has even more reader/writers for ndarrays

Structured Arrays

```
>>> x = np.zeros((2,), dtype=[('x', np.int32), \
                 ('y', np.float32), ('name', (np.string_, 10))])
>>> x[:] = [(1,2.,'Hello'),(2,3.,"World")]
>>> x
array([(1, 2.0, 'Hello'), (2, 3.0, 'World')],
     dtype=[('x', '<i4'), ('y', '<f4'), ('name', '|S10')])
>>> x[1]
(2,3.,"World")
>>> y = x['y']
>>> y
array([ 2.,  3.], dtype=float32)
>>> y[:] = 2*y
>>> y
array([ 4.,  6.], dtype=float32)
>>> x
array([(1, 4.0, 'Hello'), (2, 6.0, 'World')],
      dtype=[('x', '<i4'), ('y', '<f4'), ('name', '|S10')])
```

ndarrays can be composed of (almost) any data type. The data type is specified by the *dtype* attribute.

Note: use above dytpe instead of:
x = np.zeros((2,), dtype=[('x', 'i4'), ('y', 'f4'), ('name', 'a10')])

Copy and Referencing

```
>>> a = np.arange(12)
>>> b = a
>>> b is a
True
>>> b.shape = 3,4
>>> a.shape
(3, 4)
```

Simple assignments do NOT make a copy of *ndarray* objects (nor their associated data)

Here *a* and *b* are two names for the same *ndarray* object

Copy and Referencing

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6
>>> a.shape
(3, 4)
>>> c[0,4] = 1234
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

A shallow copy can be created with the *view* method

Here *a* and *c* are different *ndarray* objects that share the same data

Copy and Referencing

```
>>> d = a.copy()
>>> d is a
False
>>> d.base is a
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

A deep copy can be created with the *copy* method

Here a new copy of *a*'s data is created for *d*

Manipulations, Slicing, and Indexing

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000
>>> a[::-1]
array([  729,   512,   343,   216,   125, -1000,    27,
-1000,     1, -1000])
>>> for i in a:
...         print i**(1/3.),
...
nan 1.0 nan 3.0 nan 5.0 6.0 7.0 8.0 9.0
>>> b=a**(-1/3.) ; print b[0]
nan
>>> b[0] == np.nan
False
>>> np.isnan(b[0])
True
```

One dimensional *ndarray* objects can be indexed, sliced, and iterated over much like lists

Manipulations, Slicing, and Indexing

```
>>> a = np.arange(12).reshape(3,4) ; print a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = array( [ [0,1],
...              [1,2] ] )
>>> j = array( [ [2,1],
...              [3,3] ] )
>>> a[i,j]
array([[ 2,  5],
       [ 7, 11]])
>>>
>>> a[i,2]
array([[ 2,  6],
       [ 6, 10]])
>>> a.ravel()  # flatten a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
10, 11])
```

We can also give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.

stacking & concatenation

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6]])
>>> print a.shape, b.shape
(2, 2) (1, 2)
>>> x = np.concatenate((a, b), axis=0) # vertical stack
>>> print x, x.shape
[[1 2]
 [3 4]
 [5 6]] (3, 2)
>>> y = np.concatenate((a, b.T), axis=1) # horizontal
>>> print y, y.shape
[[1 2 5]
 [3 4 6]] (2, 3)
>>> np.vstack((a,b))
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.hstack((a,b.T))
array([[1, 2, 5],
       [3, 4, 6]])
```

array dimension must agree except for along the concat axis

Manipulations, Slicing, and Indexing

```
>>> a = np.array([1, 3, 0, -5, 0], float)
>>> np.where(a != 0)
(array([0, 1, 3]),)
>>> a[a != 0]
array([ 1.,  3., -5.])
>>> np.where(a != 0.0, 1 / a, a)
array([ 1.        ,  0.33333333,  0.        ,
-0.2        ,  0.        ])
>>> x = np.arange(9.).reshape(3, 3)
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
>>> np.where( x > 5 )
(array([2, 2, 2]), array([0, 1, 2]))
```

where provides a convenient (though not always fast) way to search and extract individual elements of an *ndarray* (see also *nonzero*).

Universal Functions

A universal function (or *ufunc* for short) is a function that operates on **ndarrays** in an element-by-element fashion, supporting *array broadcasting*, *type casting*, and several other standard features. That is, a ufunc is a "*vectorized*" wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs. Examples include *add, subtract, multiply, exp, log,* and *power.*

Universal Functions

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[2, 3], [4, 5]])
>>> a + b
array([[3, 5],
       [7, 9]])
>>> np.multiply(a, b)
array([[ 2,  6],
       [12, 20]])
>>> np.power(a, b)
array([[   1,    8],
       [  81, 1024]])
```

Universal functions operate on an *element-by-element* basis.

Vectorizing for Speed (I)

`numexpr`: Fast numerical array expression evaluator for Python and NumPy.

analysis of numpy string, rewrites and compiles into JIT evaluation

```
>>> a = np.arange(1e6)
>>> b = np.arange(1e6)
>>> %timeit a**2 + b**2 + 2*a*b
10 loops, best of 3: 65 ms per loop
>>> import numexpr as ne
>>> %timeit ne.evaluate("a**2 + b**2 + 2*a*b")
100 loops, best of 3: 12.6 ms per loop
>>> ne.set_num_threads(1)  # using just 1 thread
>>> %timeit ne.evaluate("a**2 + b**2 + 2*a*b")
100 loops, best of 3: 18.8 ms per loop
```

supports many common mathematical
ufuncs and `where`

http://code.google.com/p/numexpr/wiki/UsersGuide
http://www.pytables.org/docs/CISE-12-2-ScientificPro.pdf

Broadcasting

```
>>> x = np.arange(4)
>>> xx = x.reshape(4,1)
>>> y = np.ones(5)
>>> z = np.ones((3,4))
>>> x.shape
(4,)
>>> y.shape
(5,)
>>> x + y
<type 'exceptions.ValueError'>: shape mismatch:
objects cannot be broadcast to a single shape
>>> xx.shape
(4, 1)
>>> (xx + y).shape
(4, 5)
>>> xx + y
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.]])
>>> z.shape
(3, 4)
>>> (x + z).shape
(3, 4)
>>> x + z
array([[ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.]])
```

numpy will intelligently deal with *ndarrays* of different shapes. The smaller array is *broadcast* across the larger array so that they have compatible shapes

Matrix Operations

`numpy.linalg`

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[2, 3], [4, 5]])
>>> np.dot(a, b)
array([[10, 13],
       [22, 29]])
>>> np.linalg.eig(a)
(array([-0.37228132,  5.37228132]),
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
>>> np.linalg.inv(b)
array([[-2.5,  1.5],
       [ 2. , -1. ]])
```

For matrix operations, use the *numpy* methods *dot*, *cross*, and *outer*

essentially wrappers around LAPACK

Example: Chemical Stoichiometry
balance each element in chemical reactions

e.g., $aCH_4 + bO_2 \rightarrow cCO_2 + dH_2O$

a,b,c,d: "stoichiometric coefficients"

$$\begin{array}{r} \text{C} \\ \text{H} \\ \text{O} \\ \text{dummy} \end{array} \begin{pmatrix} 1 & 0 & -1 & 0 \\ 4 & 0 & 0 & -2 \\ 0 & 2 & -2 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

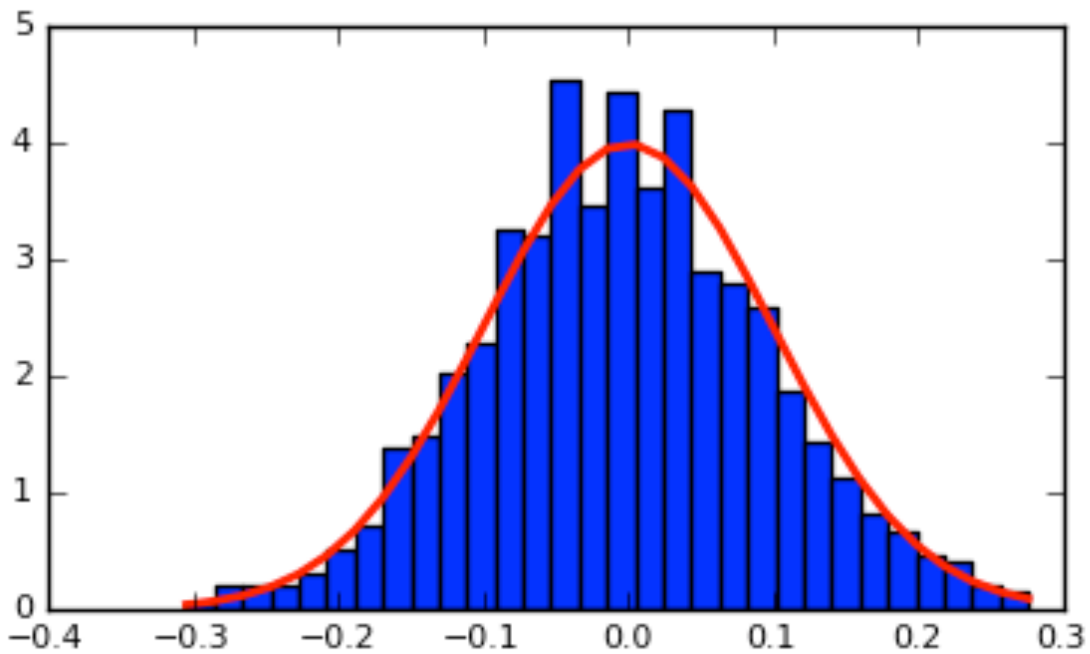
```
 >>> a = np.array([[1,0,-1,0],[4,0,0,-2],[0,2,-2,-1],
[0,0,0,1]])
>>> b = np.array([[0],[0],[0],[1]])
>>> 2*linalg.solve(a,b).T
array([[ 1.,  2.,  1.,  2.]])
```

$CH_4 + 2\ O_2 \rightarrow CO_2 + 2\ H_2O$

Random Sampling

`numpy.random`

```
>>> mu, sigma = 0, 0.1
>>> s = np.random.normal(mu, sigma, 1000)
>>> count, bins, ignored = plt.hist(s, 30,
                                    normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi))
             * np.exp( - (bins - mu)**2 /
             (2 * sigma**2) ) color='r')
>>> plt.show()
```



numpy.random most common probability density distributions, as well as a random number generator

Basic Statistics

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])
>>> np.std(a)
1.1180339887498949
>>> np.average(range(1,11), weights=range(10,0,-1))
4.0
```

Basic statistics can be calculated with built-inn *numpy* routines. More complicated tasks require *scipy*. *more later from Berian...*

Masked Arrays

```
>>> x = np.array([1, 2, 3, -1, 5])
>>> mx = np.ma.masked_array(x, mask=[0, 0, 0, 1, 0])
>>> mx.data
array([ 1,  2,  3, -1,  5])
>>> mx.mask
array([False, False, False,  True, False], dtype=bool)
>>> mx.mean()
2.75
>>> x = np.ma.array([1, 2, 3])
>>> x[0] = np.ma.masked
>>> x
masked_array(data = [-- 2 3],
             mask = [ True False False],
       fill_value = 999999)
>>> x = np.ma.array([-1, 1, 0, 2, 3], mask=[0, 0, 0, 0, 1])
>>> np.log(x)
masked_array(data = [-- 0.0 -- 0.69314718056 --],
             mask = [ True False  True False  True],
       fill_value = 1e+20)
```

MaskedArrays are a subclass of *ndarray*. In addition to standard *ndarray* properties, they contain an additional Boolean *mask* to indicate invalid data.

Where to go for help

- NumPy Tutorial:
  - http://www.scipy.org/Tentative_NumPy_Tutorial
- NumPy / SciPy documentation:
  - http://docs.scipy.org/doc/

Breakout #1

1) Make an ndarray representing a 52 deck of cards (A,2-10,J,Q,K + 4 suits) where each element represents a unique card including it's numerical equivalent (e.g., A = 1, K = 13)

```
>>> print deck[11]
('Q', 'C', 12)
```

2) Drawing 5 cards randomly from a shuffled deck, what is probability of getting at least two cards of the same value (ie. a pair)? What is the probability of getting all five cards from the same suit (ie. a flush)?

hint: np.unique, np.random.shuffle

Solutions