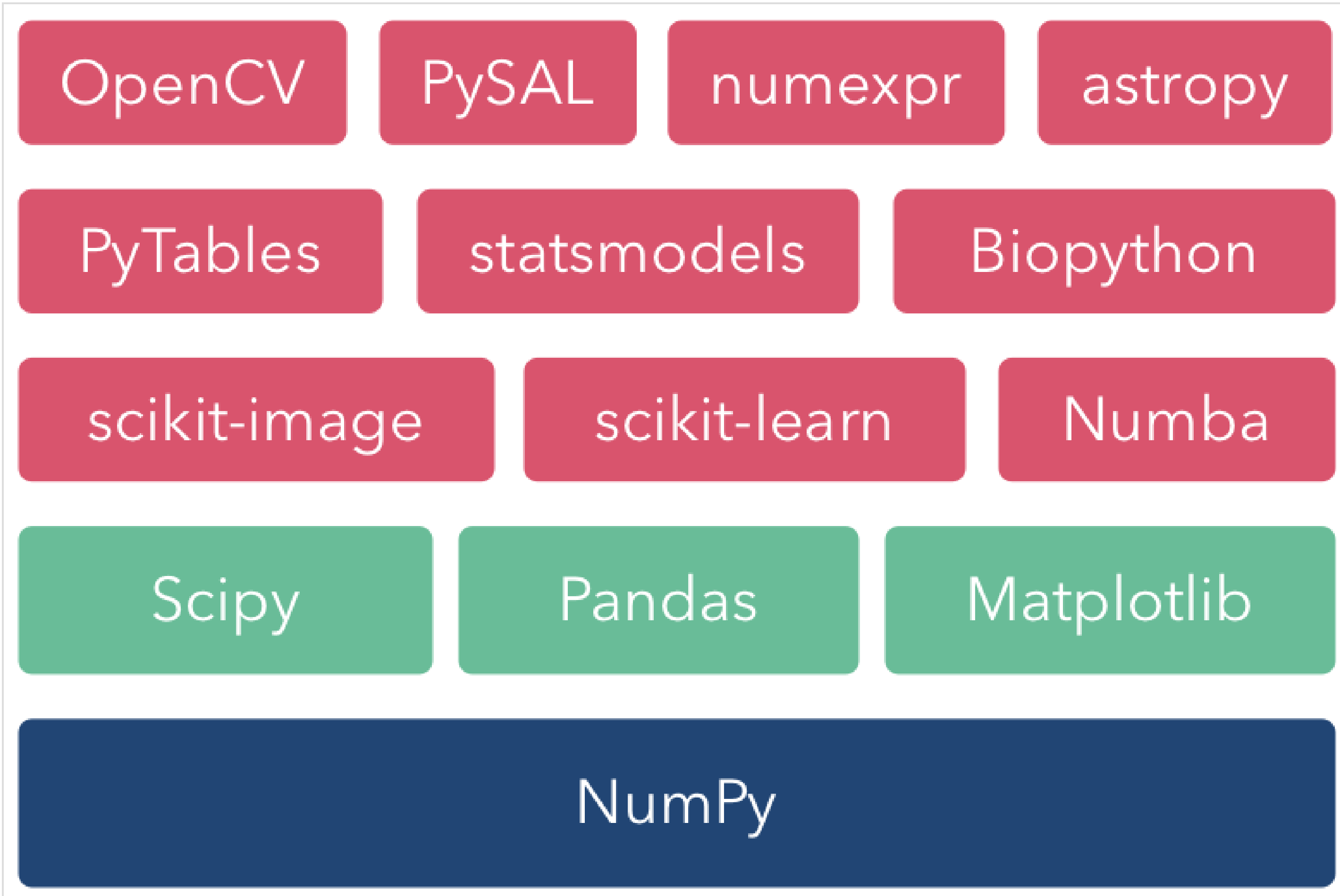
NumPy

Continuum Analytics

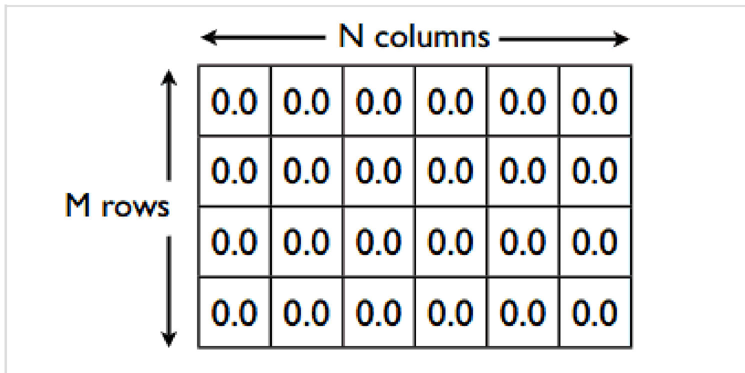
NumPy Ecosystem



Array Shape

- Arrays have a shape (dimensions)

```
import numpy as np
a = np.zeros(shape=(M,N), dtype=float)
```



- Could have fewer or more dimensions, NumPy arrays are N-dimensional

Array Shape (cont.)

- 1 dimensional array

```
>>> np.zeros(shape=5, dtype='float')
array([ 0.,  0.,  0.,  0.,  0.])
```

- 3 dimensional array

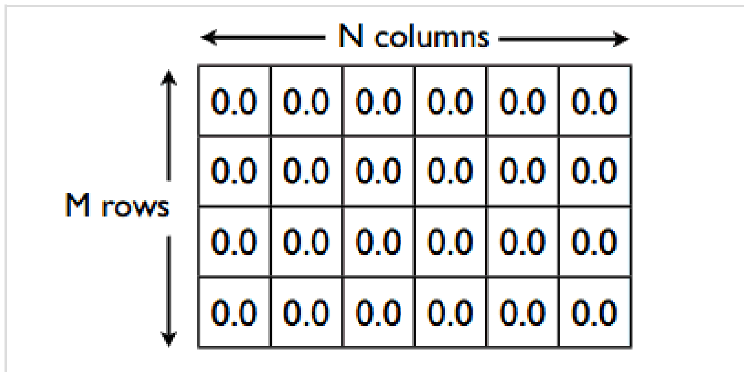
```
>>> np.zeros(shape=(3,3,3), dtype='float')
    array([[[ 0.,  0.,  0.],
            [ 0.,  0.,  0.],
            [ 0.,  0.,  0.]],
           [[ 0.,  0.,  0.],
            [ 0.,  0.,  0.],
            [ 0.,  0.,  0.]],
           [[ 0.,  0.,  0.],
            [ 0.,  0.,  0.],
            [ 0.,  0.,  0.]]])
```

CONTINUUM ANALYTICS

Array Type

- Arrays have a specified dtype

```python
import numpy as np
a = np.zeros(shape=(M,N), dtype=float)
```



- Every element must be of that type (but types can be records)

Array Creation

- You can create arrays from python lists

```
>>> np.array([[1,2,3], [4,5,6]], dtype='float')
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

- Creating NumPy arrays from lists is not very efficient, native python data types are slow
- Often read and write directly from files instead
- Or use some other utility, `zeros()`, `diag()`, `ones()`, `arange()`

Array Creation (cont.)

- Evenly spaced values on an interval
    - `arange([start,] end, [,step])`

```
>>> np.arange(0, 5)
array([0, 1, 2, 3, 4])

>>> np.arange(0, 5, 0.5)
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,
   4. ,  4.5])

>>> np.arange(10, 0, -1)
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

- `arange` allows fractional and negative steps

Array Creation (cont.)

- Values equidistant on a linear scale
  - `linspace(start, end, num)`

```
>>> np.linspace(0, 5, 11)
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,
        4. ,  4.5,  5. ])
```

- Values equidistant on a log scale
  - `logspace(start, stop, num)`

```
>>> np.logspace(0, 3, 4)
array([    1.,    10.,   100.,  1000.])
```

- Note: end-points by default included (use `num=N+1` for N segments)

Array Creation (cont.)

- Zero-initialized

```
>>> np.zeros((2,2))
array([[ 0.,  0.],
       [ 0.,  0.]])
```

- One-initialized

```
>>> np.ones((1,5))
array([[ 1.,  1.,  1.,  1.,  1.]])
```

- Uninitialized

```
>>> np.empty((1,3))
array([[  2.12716633e-314,   2.12716633e-314,   2.1520
3762e-314]])
```

Array Creation (cont.)

- Constant diagonal value

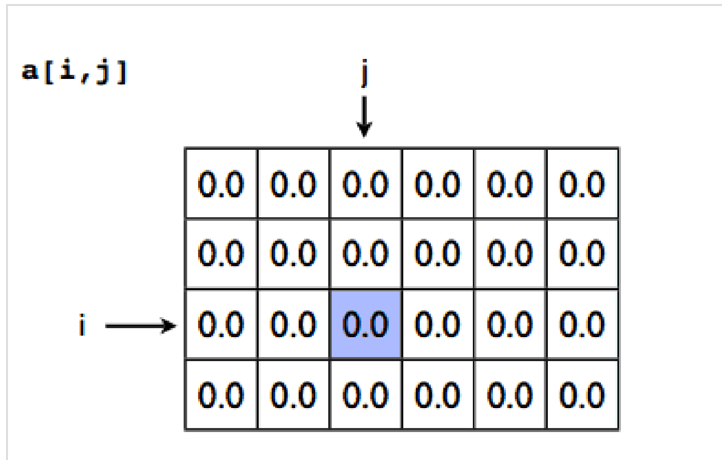
```
>>> np.eye(3)
array([[ 1.,   0.,   0.],
       [ 0.,   1.,   0.],
       [ 0.,   0.,   1.]])
```

- Multiple diagonal values

```
>>> np.diag([1,2,3,4])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Array Access (items)

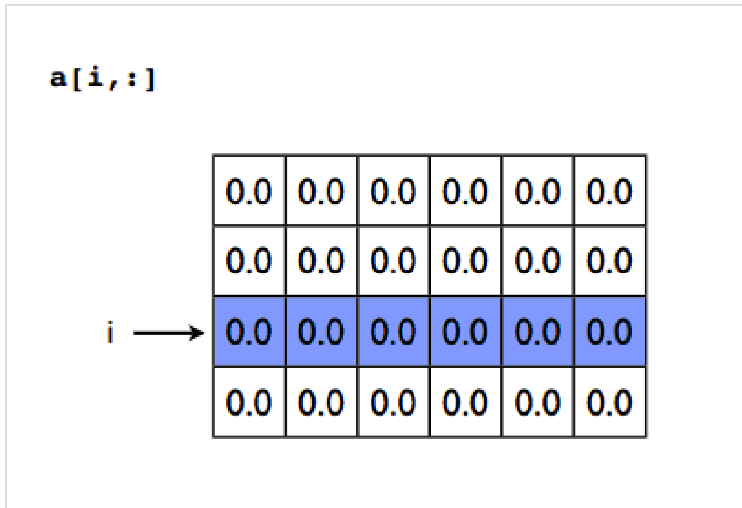
- Accessing single item: `arr[row, column]`



- Syntax is different from accessing nested lists (`lst[i][j]`)

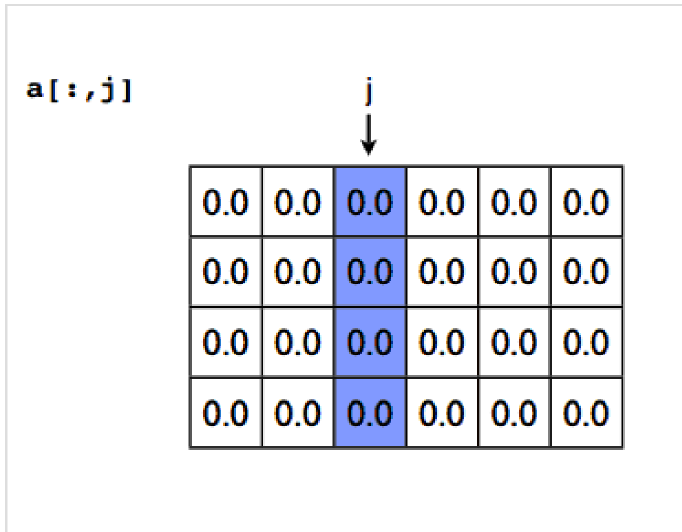
Array Access (rows)

- Accessing the entire row: `arr[row, :]`



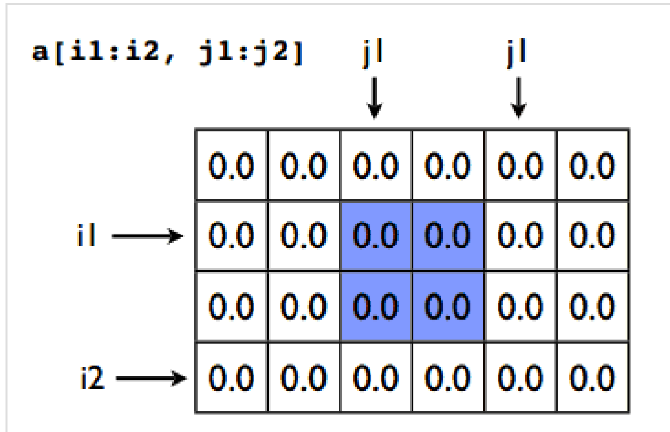
Array Access (columns)

- Accessing the entire column: `arr[:, column]`



Array Access (regions)

- Accessing a region (slice)



- Standard slicing syntax applies `(start:end:stride)`

Array Slicing

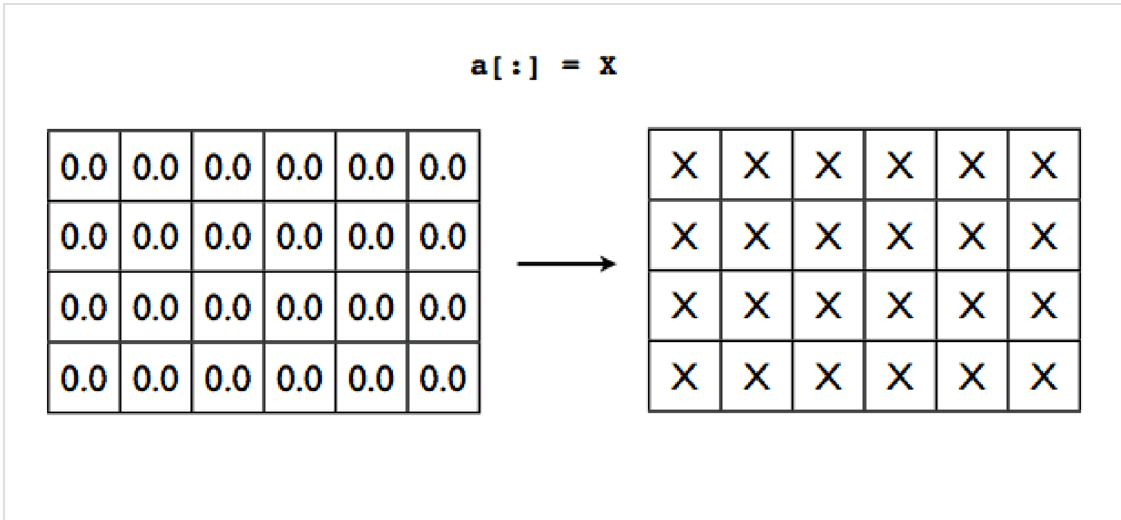
- Slices are never copies

```
>>> a = np.arange(0, 5, 0.5)

>>> a
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,
  4. ,  4.5])

>>> a[::2]
array([ 0.,  1.,  2.,  3.,  4.])

>>> a[::2][0] = 999

>>> a
array([ 999. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.
5,  4. ,  4.5])
```

- Having shared data is different from list slicing
    - saves memory and makes operations more efficient

Array Assignment

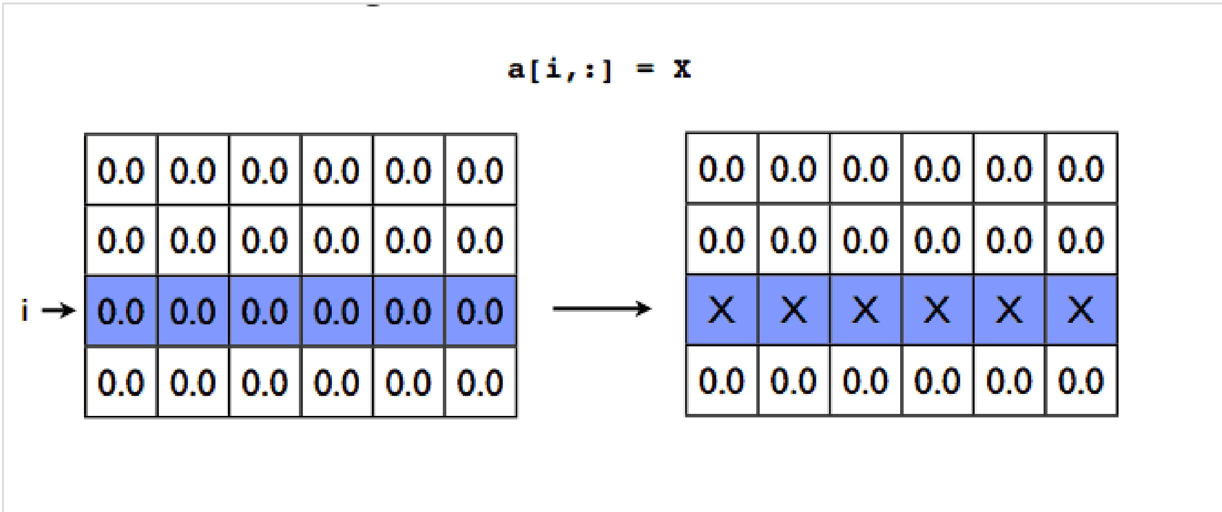
- Assign all elements to a scalar



This "copying" of the scalar, as needed, is an example of NumPy broadcasting (more later).

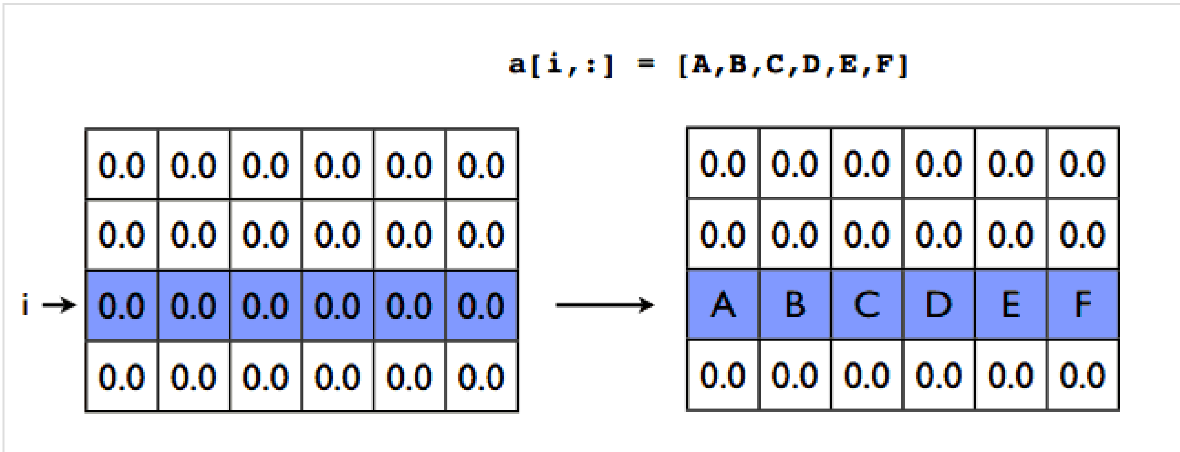
Array Assignment (cont.)

- Row or column assignment to a scalar



Array Assignment (cont.)

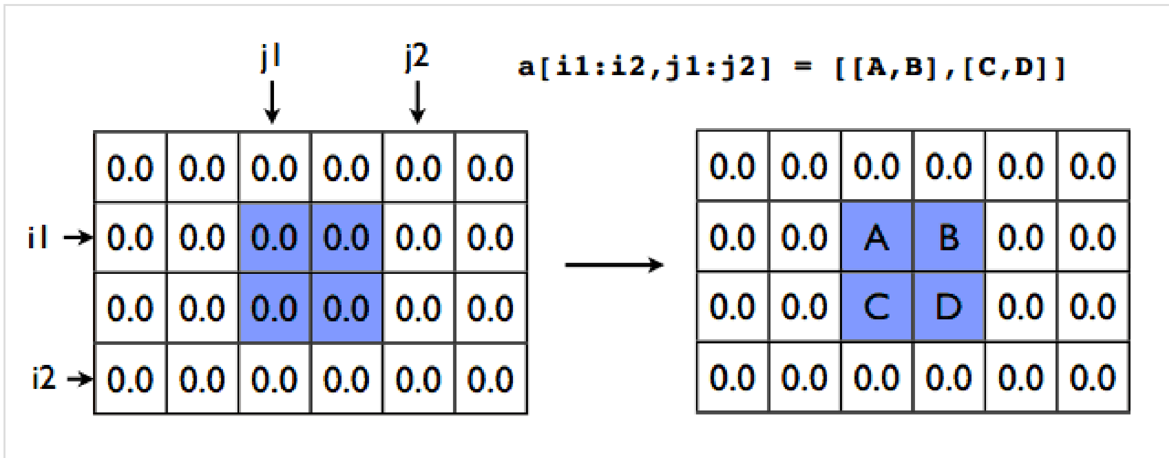
- Row or column assignment to a sequence



- Shapes must match

Array Assignment (cont.)

- You can also do similar assignments to regions



- Again, shapes must match

Common Slicing Patterns

- Shifting values of an array: `a[1:] = a[:-1]`

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a[1:] = a[:-1]

>>> a
array([0, 0, 1, 2, 3, 4, 5, 6, 7, 8])
```

- Reverse an array: `a[::-1]`

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

CONTINUUM ANALYTICS

Array Math

- Operations with scalars apply to all elements

```
>>> a = np.arange(10)

>>> a + 20
array([20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
```

- Operations on other arrays are element-wise

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a + a
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

- These operations create new arrays for the result

Vectorized Conditionals

- Conditional operations make boolean arrays

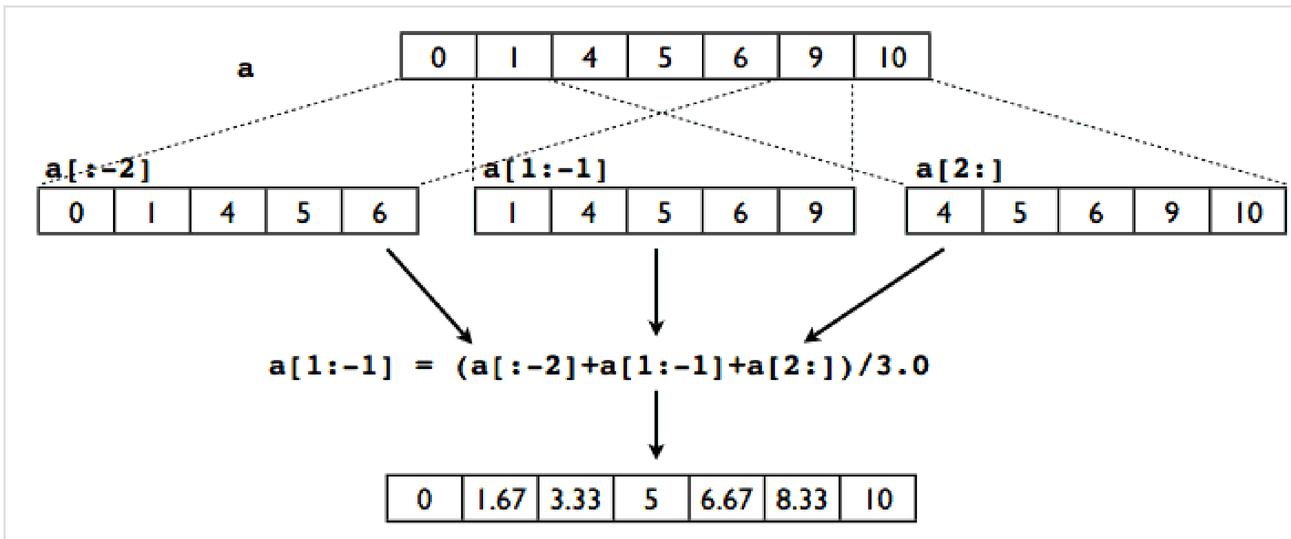
```
>>> a
array([0, 1, 2, 3, 4, 5])

>>> a > 2
array([False, False, False, True,  True,  True], dtype
=bool)
```

- `np.where` selects from an array

```
>>> a
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

>>> np.where(a > 15)
(array([6, 7, 8, 9]),)

>>> np.where(a > 15, a, False)
array([ 0,  0,  0,  0,  0,  0, 16, 17, 18, 19])
```

Operations on Grids of Data

- Slicing and array operations can sometimes be used for discretized grids



- May need padding (and care!) to meet appropriate boundary conditions

Array Methods

- Predicates
  - `a.any(), a.all()`
- Reductions
  - `a.mean(), a.argmin(), a.argmax(), a.trace(), a.cumsum(), a.cumprod()`
- Manipulation
  - `a.argsort(), a.transpose(), a.reshape(...), a.ravel(), a.fill(...), a.clip(...)`
- Complex Numbers
  - `a.real, a.imag, a.conj()`

CONTINUUM ANALYTICS

Reshaping Arrays

The size of an array is the product of the shape values: an array with a shape of `(3,4,2)` has `3x4x2=24` elements in it.

- `reshape` will reshape an array to any matching size
- Takes tuples as arguments: `a.reshape((3,1,1,1))`
- Use `-1` for a wildcard dimension
    - gets filled in based on the shape of the array

```
>>> a = np.arange(30)
>>> b = a.reshape((3,-1,2))
>>> b.shape
(3, 5, 2) #because 30 / (3 * 2) = 5!!
```

Reshaping Arrays (cont.)

- `np.ravel` and `np.flatten` reshape arrays to one dimension

```
>>> b.shape
(3, 5, 2)
>>> b.flatten().shape
(30,)
```

The `flatten` function always makes a copy of the data, `ravel` only does so if it is necessary.

Reshaping Arrays (cont.)

- `np.squeeze` removes singular dimensions

```
>>> c = b.reshape((3,1,5,1,2))
>>> c.shape
(3, 1, 5, 1, 2)
>>> np.squeeze(c).shape
(3, 5, 2)
```

This situation happens when a selection operation eliminates all of one dimension.

Array dtype

- NumPy array elements have a single data type
- The type object is accessible through the `.dtype` attribute

Most important attributes of `dtype` objects:

- `dtype.byteorder` — big or little endian
- `dtype.itemsize` — element size of this dtype
- `dtype.name` — a name for this dtype object
- `dtype.type` — type object used to create scalars

There are many others...

Array dtype (cont.)

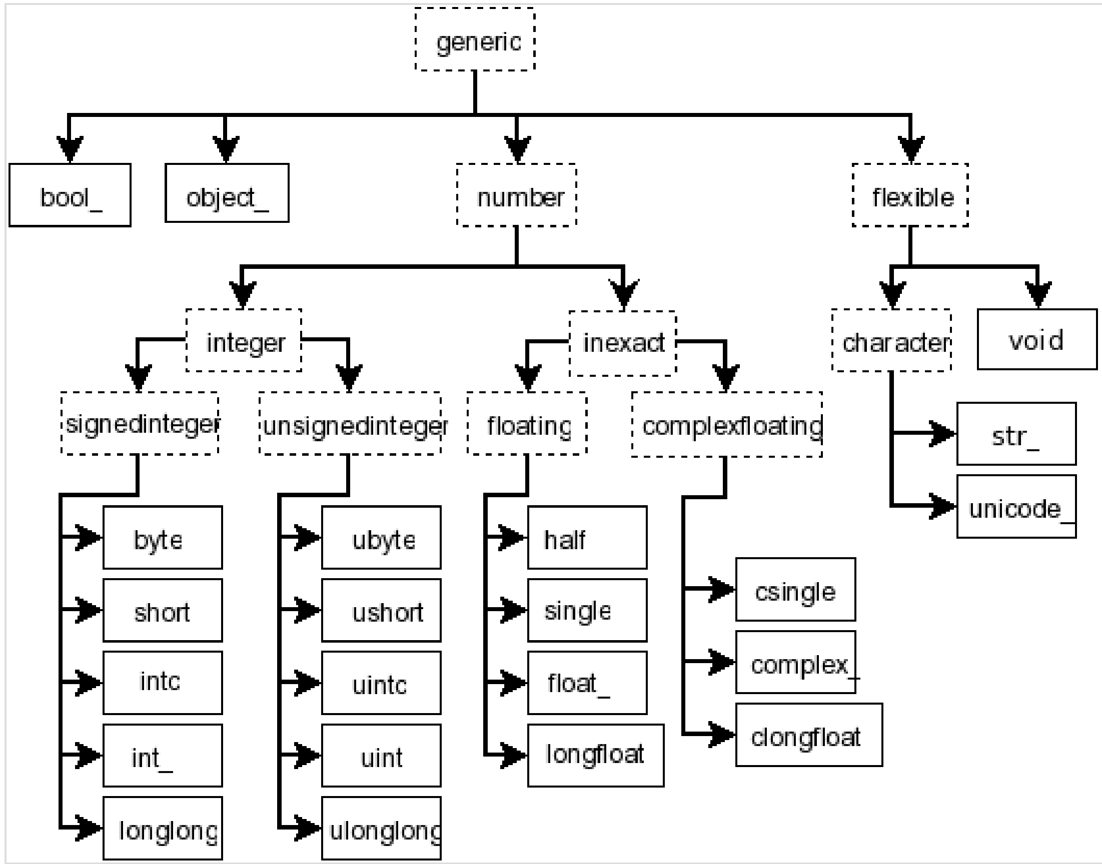
- Array dtypes are usually inferred automatically

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

>>> a.dtype
dtype('int64')

>>> b = np.array([1,2,3,4.567])

>>> b.dtype
dtype('float64')
```

- But can also be specified explicitly

```
>>> a = np.array([1,2,3], dtype=np.float32)

>>> a.dtype
dtype('int64')

>>> a
array([ 1.,  2.,  3.], dtype=float32)
```

Hierarchy of dtypes



`np.datetime64` is a new addition in NumPy 1.7

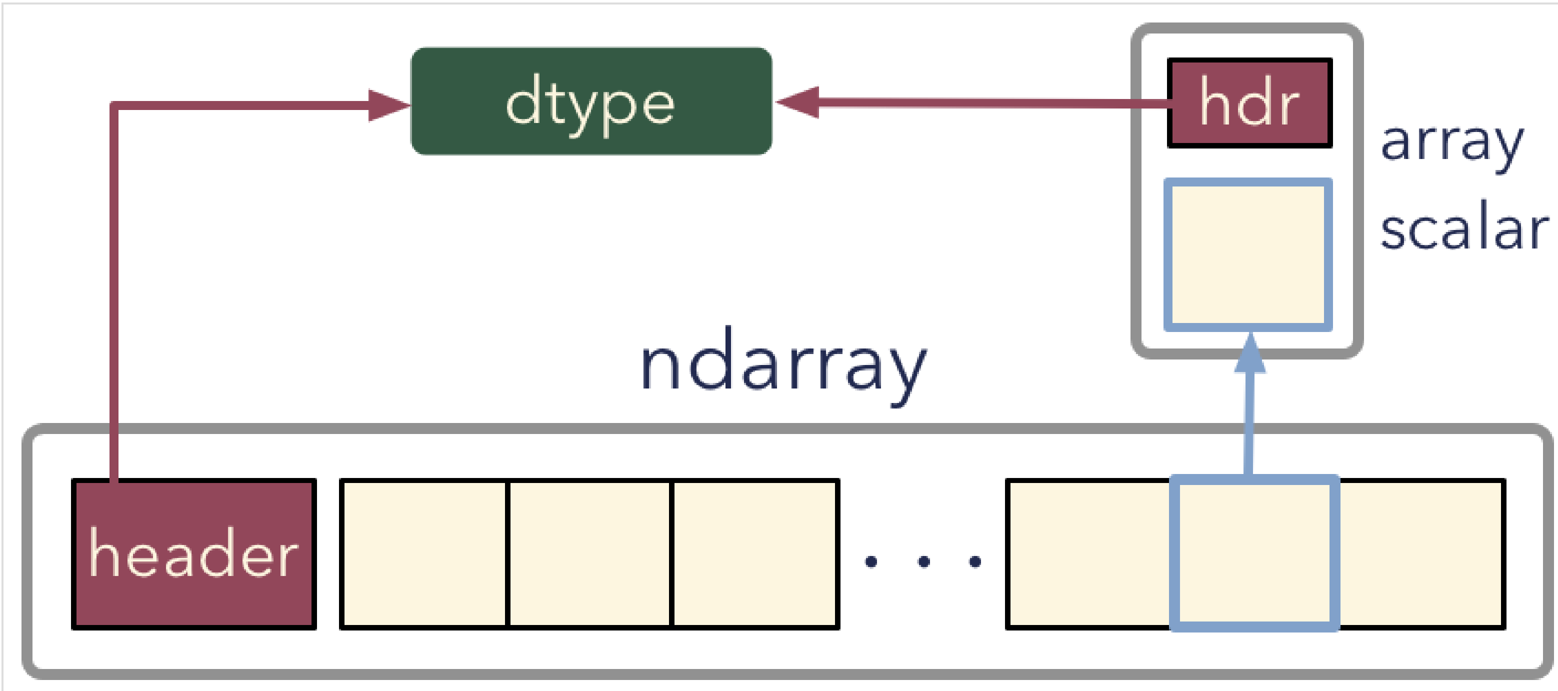
NumPy Data Model

- Metadata — dtype, shape, strides
- Memory pointer to data

When slicing arrays, NumPy creates new dtype/shape/stride information, but reuses the pointer to the original data.

```
>>> a = np.arange(10)
>>> b = a[3:7]
>>> b
array([3, 4, 5, 6])

>>> b[:] = 0
>>> a
array([0, 1, 3, 0, 0, 0, 0, 7, 8, 9])

>>> b.flags.owndata
False
```

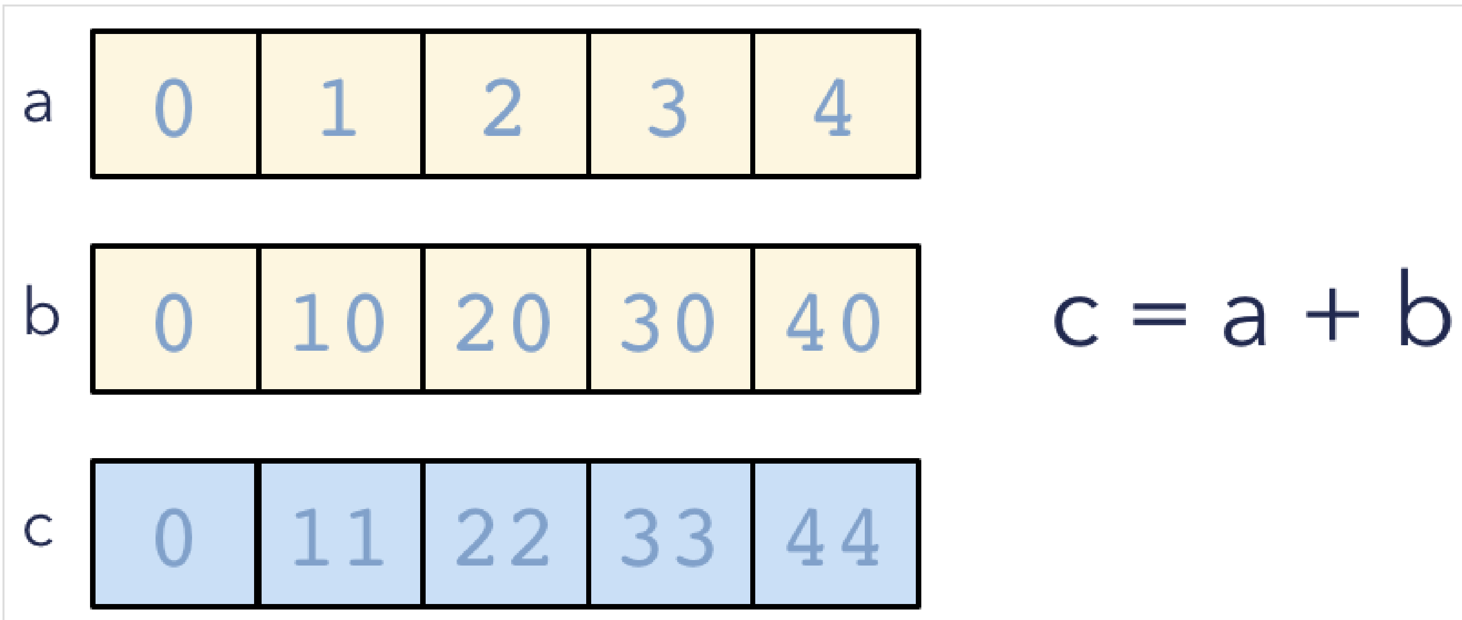
Array Memory Layout



- Array scalars are lightweight wrappers that adapt single scalar objects to behave as a single- element array

Universal Functions (ufuncs)

NumPy ufuncs are functions that operate element-wise on one or more arrays.



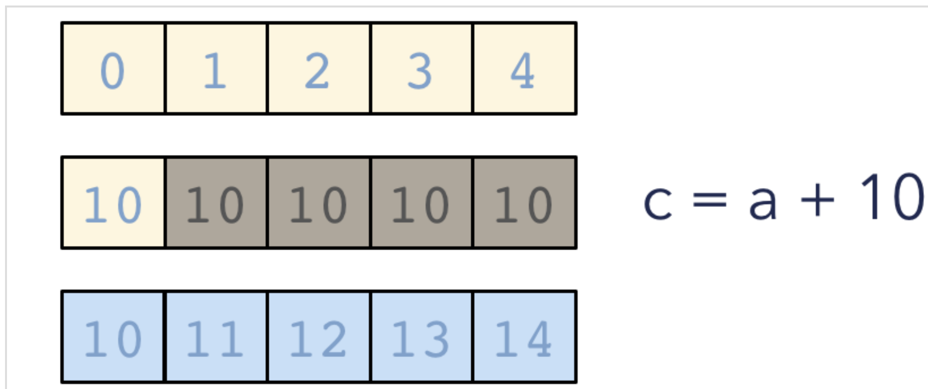
When called, ufuncs dispatch to optimized C inner-loops based on array dtype.

NumPy has many built-in ufuncs

- comparison: <, <=, ==, !=, >=, >
- arithmetic: +, -, *, /, reciprocal, square
- exponential: exp, expm1, exp2, log, log10, log1p, log2, power, sqrt
- trig: sin, cos, tan, acsin, arccos, atctan, sinh, cosh, tanh, acsinh, arccosh, atctanh
- bitwise: &, |, ~, ^, left*shift, right*shift
- logical operations: and, logical_xor, not, or
- predicates: isfinite, isinf, isnan, signbit
- other: abs, ceil, floor, mod, modf, round, sinc, sign, trunc

Broadcasting

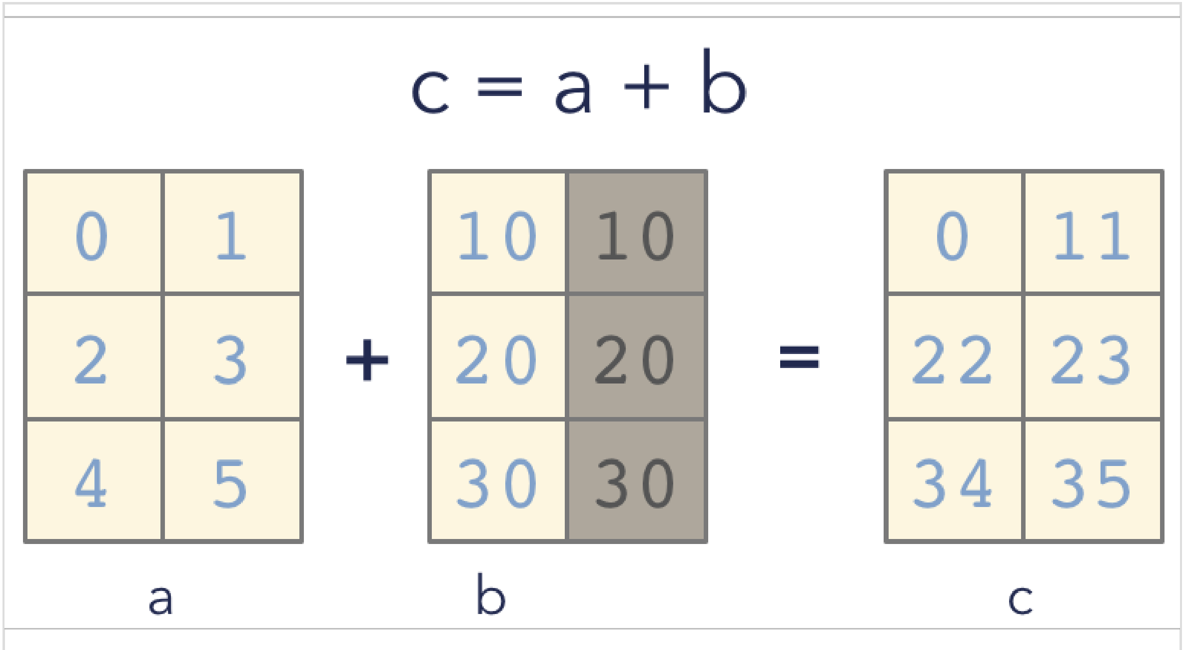
- Broadcasting is a key feature of NumPy
  - arrays with different, but compatible shapes can be used as arguments to ufuncs



- Here, a scalar 10 is broadcast to an array with shape `(5,)`

Broadcasting (cont.)

More involved broadcasting example in two dimensions



Here, an array of shape `(3, 1)` is broadcast to an array with shape `(3, 2)`

Broadcasting Rules

In order for an operation to broadcast, the size of all the trailing dimensions for both arrays must either:

be **equal** OR be **one**

```
A      (1d array):                  3
B      (2d array):              2 x 3
Result (2d array):              2 x 3

A      (2d array):              6 x 1
B      (3d array):          1 x 6 x 4
Result (3d array):          1 x 6 x 4

A      (4d array):      3 x 1 x 6 x 1
B      (3d array):          2 x 1 x 4
Result (4d array):      3 x 2 x 6 x 4
```

Square Peg in a Round Hole

If the dimensions do not match up, `np.newaxis` may be useful.

```
>>> a = np.arange(6).reshape((2, 3))

>>> b = np.array([10, 100])

>>> a * b
-------------------------------------------------------------------------
ValueError                                          Traceback (most recent call last)
<ipython-input-18-50927f39610b> in <module>()
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (2,3) (2)

>>> b[:,np.newaxis].shape
(2, 1)

>>> a *b[:,np.newaxis]
array([[  0,  10,  20],
       [300, 400, 500]])
```

CONTINUUM ANALYTICS

Fancy Indexing

NumPy arrays may be used to index into other arrays.

```
>>> a = np.arange(15).reshape((3,5))

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

>>> i = np.array([[0,1], [1, 2]])

>>> j = np.array([[2, 1], [4, 4]])

>>> a[i,j]
array([[ 2,  6],
       [ 9, 14]])
```

Fancy Indexing (cont.)

Boolean arrays can also be used to index into other arrays.

```
>>> a = np.arange(15).reshape((3,5))

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

>>> b = (a % 3 == 0)

>>> b
array([[ True, False, False,  True, False],
       [False,  True, False, False,  True],
       [False, False,  True, False, False]],
       dtype=bool)

>>> a[b]
array([ 0,  3,  6,  9, 12])
```

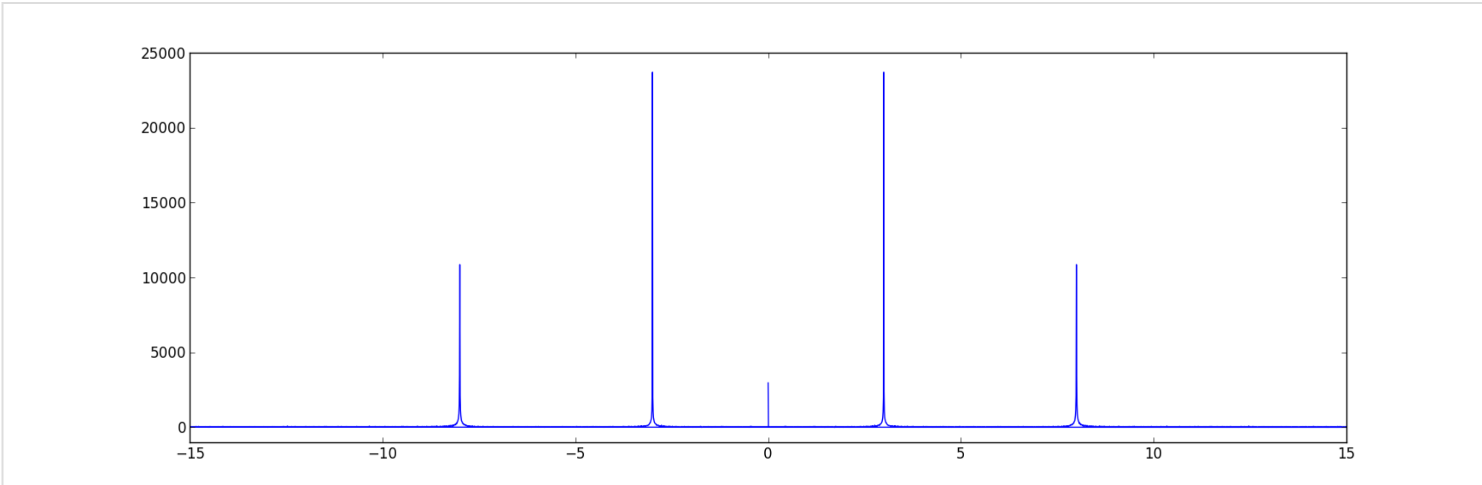
Other Subpackages

- `numpy.fft` — Fast Fourier transforms
- `numpy.polynomial` — Orthogonal polynomials, spline fitting
- `numpy.linalg` — Linear algebra
  - `cholesky, det, eig, eigvals, inv, lstsq, norm, qr, svd`
- `numpy.math` — C standard library math functions
- `numpy.random` — Random number generation
  - `beta, gamma, geometric, hypergeometric, lognormal, normal, poisson, uniform, weibull`

`numpy.fft` Example

```
>>> import numpy as np
>>> PI = np.pi
>>> t = np.linspace(0,120,4000)
>>> signal = 12*np.sin(3 * 2*PI*t)         # 3 Hz
>>> signal += 6*np.sin(8 * 2*PI*t)         # 8 Hz
>>> signal += 1.5*np.random.random(len(t)) # noise
>>> FFT = abs(np.fft.fft(signal))
>>> freqs = np.fft.fftfreq(signal.size, t[1]-t[0])
```

`numpy.fft` (Cont..)



`numpy.random` Example

```
>>> np.random.random((4,5))
array([[ 0.55761998,  0.40232521,  0.51377386,  0.4662
7006,  0.56271765],
       [ 0.08310412,  0.5209415 ,  0.01124337,  0.7675
9096,  0.11065584],
       [ 0.14694165,  0.44694399,  0.03155987,  0.8372
7212,  0.32643212],
       [ 0.86353036,  0.41302666,  0.86132392,  0.0625
5084,  0.11043359]])

>>> np.random.normal(loc=0, scale=2, (10,))
array([ 0.1553377 , -1.21074194, -2.50073224, -0.42407
267, -1.51815018,
        1.81386522,  1.55730186, -4.64003269,  0.73482
302,  2.85621511])

>>> np.random.hypergeometric(20, 10, 5, (10,))
array([3, 4, 3, 4, 4, 2, 3, 5, 4, 3])

>>> np.random.permutation(np.arange(10))
array([6, 8, 1, 4, 3, 7, 0, 5, 9, 2])
```

`numpy.linalg` Example

```
>>> a = np.array([[2,1,1], [1,3,2], [-1,1,2]])

>>> a.T
array([[ 2,  1, -1],
       [ 1,  3,  1],
       [ 1,  2,  2]])

>>> x = np.array([10, 10, 10])

>>> y = np.array([40, 60, 20])

>>> np.linalg.inv(a)
array([[ 0.5  , -0.125, -0.125],
       [-0.5  ,  0.625, -0.375],
       [ 0.5  , -0.375,  0.625]])

>>> np.dot(a, x)
array([40, 60, 20])

>>> np.linalg.solve(a,y)
array([ 10.,  10.,  10.])
```

Array Subclasses

- `numpy.matrix` — Matrix operators
- `numpy.recarray` — Record arrays
- `numpy.ma` — Masked arrays
- `numpy.memmap` — Memory-mapped arrays

`numpy.matrix` Example

NumPy matrix objects can be created using matlab-like syntax:

```
>>> from numpy import matrix

>>> A = matrix('1.0 2.0; 3.0 4.0')

>>> A
matrix([[ 1.,  2.],
        [ 3.,  4.]])

>>> Y = matrix('5.0; 7.0')

>>> Y
matrix([[ 5.],
        [ 7.]])
```

`numpy.matrix` Example (cont.)

Multiplication is "matrix multiplication", instead of being element-wise:

```
>>> from numpy.linalg import solve

>>> X = solve(A, Y)

>>> X
matrix([[-3.],
        [ 4.]])

>>> A * X
matrix([[ 5.],
        [ 7.]])

>>> A.I
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

Array I/O

- csv text files
  - `genfromtxt`, `loadtxt`, `savetxt`, `fromfile`
  - `loadtxt` is a simpler interface for `genfromtxt`
  - `np.loadtxt(fname, dtype, delimiter)`

Array I/O (Cont...)

- NPZ (pickled NumPy objects)
  - `load`, `save`, `savez`

```
>>>d = StringIO("M 21 72\nF 35 58")

>>>data = np.loadtxt(d, dtype={'names': ('gender', 'ag
e', 'weight'),
... 'formats': ('S1', 'i4', 'f4')})

>>>print data
array([('M', 21, 72.0), ('F', 35, 58.0)],
      dtype=[('gender', 'S1'), ('age', 'i4'), ('weight
', 'f4')])

>>>np.savez('data.npz',data)
```