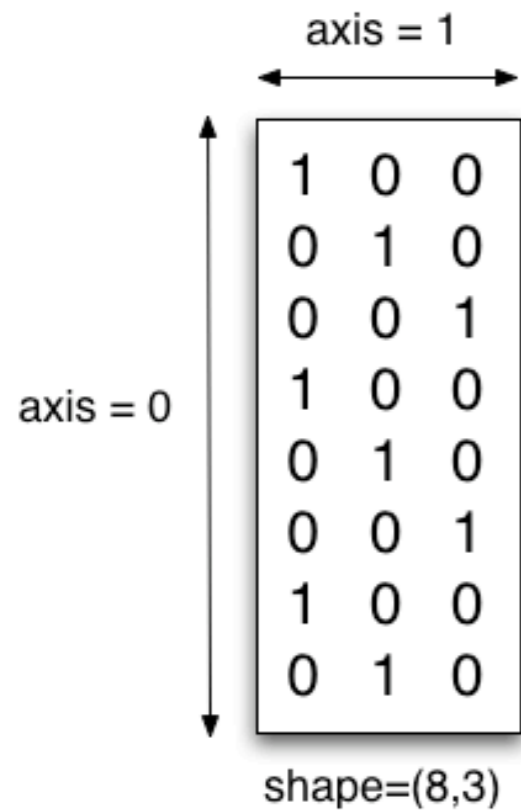


SciPy

- **scipy** [www.scipy.org and links on course web page]
 - scipy is a collection of many useful numerical algorithms (numpy is the array core)
 - Python wrappers around compiled libraries and subroutines (Fortran, C)
- **scipy arrays**
 - like built-in Python lists, except scipy arrays:
 - ▶ are multidimensional and of rectangular shape (not lists of lists)
 - ▶ have elements of homogeneous types, not arbitrary collections
 - ▶ support “array syntax”, i.e., aggregate operations on arrays
 - ▶ support slicing across all axes
 - ▶ are more efficient to manipulate (looping in C, not Python)
 - more like arrays/matrices in Matlab

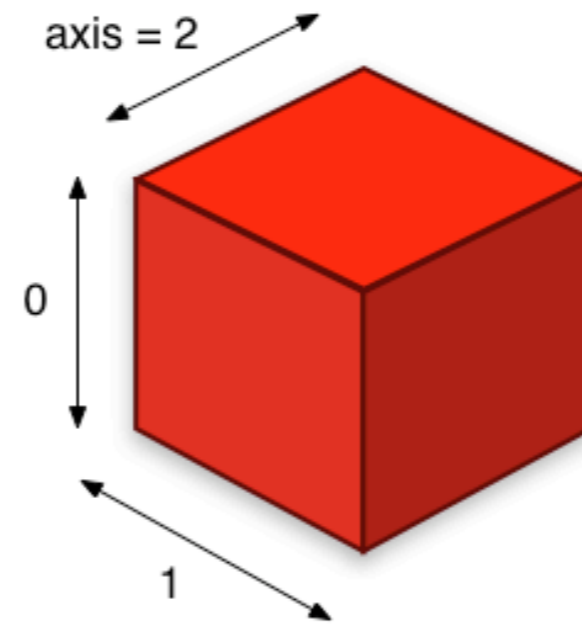
scipy arrays

Anatomy of an array



The **axes** of an array describe the order of indexing into the array, e.g., axis=0 refers to the first index coordinate, axis=1 the second, etc.

The **shape** of an array is a tuple indicating the number of elements along each axis. An existing array **a** has an attribute **a.shape** which contains this tuple.



- all elements must be of the same dtype (datatype)
- the default dtype is float
- arrays constructed from list of mixed dtype will be upcast to the "greatest" common type

Constructing arrays

- **scipy.array(alist)**: construct an n-dimensional array from a Python list
 - `a = scipy.array([[1,2,3],[4,5,6]])`
 - `b = scipy.array([i*i for i in range(100) if i%2==1])`
 - `c = b.tolist()` # convert array back to Python list
- **scipy.zeros(shape, dtype=float)**: construct an n-dimensional array of the specified shape, filled with zeros of the specified type
 - `a = scipy.zeros(100)` # 100-element array of float zeros
 - `b = scipy.zeros((2,8), int)` # 2x8 array of int zeros
 - `c = scipy.zeros((N,M,L), complex)` # NxMxL array of 0.+0.j
- **scipy.ones(shape, dtype=float)**: construct an n-dimensional array of the specified shape, filled with ones of the specified type
 - `a = scipy.ones(10, int)` # 10-element array of int ones
 - `b = scipy.pi * scipy.ones(5,5)` # 5x5 array of pi's

Constructing arrays (continued)

- `scipy.eye(N, M, dtype=float)`: construct a 2D NxM identity matrix
 - `identity = scipy.eye(10,10, float)`
 - `offdiag = scipy.eye(10,10,1)+scipy.eye(10,10,-1)`
- `scipy.transpose(a)`
 - `b = scipy.transpose(a)` # rev. dim. of a (even for dim > 2)
 - `b = a.T` # same as `scipy.transpose(a)`
 - `c = scipy.swapaxes(a, axis1, axis2)` # swap specified axes
- `scipy.arange` and `scipy.linspace`
 - `a = scipy.arange(start, stop, increment)` # like `range()`, but with potentially real-valued increment
 - `b = scipy.linspace(start, stop, num_elements)` # specify number of points (and whether or not endpoints are included)

Random arrays

- `scipy.random.random(shape):` # uniform random on [0.,1.)
 - `scipy.random.random((100,100))`
 - `scipy.pi * scipy.random.random(1000)`
- `scipy.random.randint(lo, hi, shape):` # ints on [lo, hi)
 - `scipy.random.randint(0,2,100)` # binary array of length 100
 - `scipy.random.randint(1,10, (10,10))`
- `scipy.random.standard_normal(shape)` # mean=0, std=1 Gaussian
 - `scipy.random.standard_normal((5,10,15))`
 - `10. + 3.*scipy.random.standard_normal(100)` # mean=10, std=3

Indexing arrays

- multidimensional indexing
 - `elem = a[i,j,k]` # `a[i][j][k]` less efficient
- negative indexing: wrap around end of array
 - `last_elem = a[-1]`
- arrays as indices
 - `i = scipy.array([0,1,2,1])`
 - `j = scipy.array([1,2,3,4])`
 - `a[i,j] -> array(a[0,1], a[1,2], a[2,3], a[1,4])`
 - `b = scipy.array([True, False, True, False])`
 - `a[b] -> array(a[0], a[2])` since only `b[0]` and `b[2]` True

Slicing arrays (extracting subsections)

- slice a defined subblock
 - `section = a[10:20, 30:40]` # 10x10 block from [10,30]
 - `everyother = a[10:20:2, 30:40:2]` # 5x5 striped block
- grab everything up to the beginning/end of array
 - `asection = a[10:, 30:]` # missing stop -> until end of array
 - `bsection = b[:10, :30]` # missing start -> until start

Slicing arrays (extracting subsections)

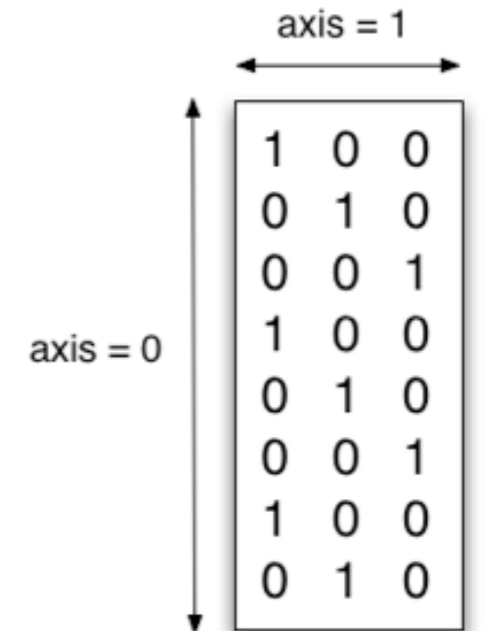
- grab an entire column

- `x = a[:,0]` # everything in the 0th column (no start, stop)

- `y = a[:,1]` # everything in 1st column

- `z = a[:,2]` # everything in second column

↑
slice from start to stop along axis=0

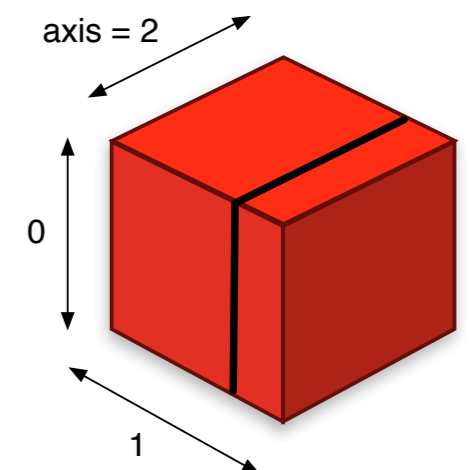


- slice of an end of array

- `tail = a[-10:]` # last 10 elements of array

- `interior = c[1:-1, 1:-1, 1:-1]` # everything but outer shell

- `slab = b[:, -10: :, :]` # slab of width 10 off side of array



Array syntax: element-wise operations

- arithmetic & trigonometric operations (efficient syntactically and numerically - no looping)
 - `c = a + b` # element-wise sum (a, b must be same shape)
 - `d = e * f` # element-wise mul (NOT matrix multiplication)
 - `g = -h` # negate every element
 - `y = (x+1)%2` # swap 0's and 1's in binary array
 - `logspace = 10.**scipy.linspace(-6.0, -1.0, 50)`
 - # 50 equally-spaced-in-log points from 1.e-6 to 1.e-1
 - `y = scipy.sin(x)`
 - `w = scipy.exp((0.+1.j)*theta)`
- logical operations
 - `z = w > 0.0` # bool array indicating which elements > 0.0 (same shape as w)

Functions on arrays: sums, etc.

- simple sums

- `s = scipy.sum(a)`

- `s0 = scipy.sum(a, axis=0)` # sum along axis 0, return array of reduced shape

- averaging

- `m = scipy.mean(a, axis)` # mean along axis; over entire array if axis not specified (`axis=None`)

- `s = scipy.std(a, axis)` # std. dev. along axis

- cumulative sums

- `s0 = scipy.cumsum(a, axis=0)`

- `s = scipy.cumsum(a)` # if no axis given, unravel entire array and return 1-d cumulative sum

More functions on arrays (a few of many)

- **scipy.any(a)**: return True if any element of a is True
- **scipy.all(a)**: return True if all elements of a are True
- **scipy.alltrue(a, axis)**: perform logical_and along given axis of a
- **scipy.append(a, values, axis)**: append values to a along specified axis
- **scipy.concatenate((a1, a2, ...), axis)**: concatenate tuple of arrays along specified axis
- **scipy.min(a, axis=None), scipy.max(a, axis=None)**: get min/max values of a along specified axis (global min/max if axis=None)
- **scipy.argmin(a, axis=None), scipy.argmax(a, axis=None)**: get indices of min/max of a along specified axis (global min/max if axis=None)
- **scipy.reshape(a, newshape)**: reshape a to newshape (must conserve total number of elements)
- **scipy.matrix(a)**: create matrix from 2D array a (matrices implement matrix multiplication rather than element-wise multiplication)
- **scipy.histogram, scipy.histogram2d, scipy.histogramdd**: 1-dimensional, 2-dimensional, and d-dimensional histograms, respectively
- **scipy.round(a, decimals=0)**: round elements of matrix a to specified number of decimals
- **scipy.sign(a)**: return array of same shape as a, with -1 where $a < 0$, 0 where $a = 0$, and +1 where $a > 0$
- **a.tofile(fid, sep="", format="%s")**: write a to specified file (fid), in either binary or ascii format depending on options
- **scipy.fromfile(file=, dtype=float, count=-1, sep='')**: read array from specified file (binary or ascii)
- **scipy.unique(a)**: return sorted unique elements of array a
- **scipy.where(condition, x, y)**: return array with same shape as condition, where values from x are inserted in positions where condition is True, and values from y where condition is False

Numerical methods in scipy

- linear algebra

```
import scipy.linalg

# linear algebra routines in scipy.linalg module

inv_m = scipy.linalg.inv(m)
det_m = scipy.linalg.det(m)
sol_ab = scipy.linalg.solve(a,b) # solve a.x = b

eigenvals, eigenvecs = scipy.linalg.eig(m)
u, sigma, vH = scipy.linalg.svd(m) # singular value decomp.

m = scipy.matrix([[1,2,3],[4,5,6],[7,8,9]])
n = scipy.matrix([[0,1,0],[1,1,1],[1,0,0]])
matrxiproduct = m * n # matrix multiplication (NOT element-wise)

# etc.
```

Numerical methods (continued)

- ODE integration: arrays as the lingua franca of scipy

```
import scipy, scipy.integrate      # import both the top-level scipy
                                    # namespace, and the lower-level
                                    # scipy.integrate module

def Lorenz(w,t,S,R,B):              # define a right-hand side function
    x,y,z = w
    return scipy.array([S*(y-x), R*x-y-x*z, x*y-B*z])

w_initial = scipy.array([0.,1.,0.])
timepoints = scipy.linspace(0., 100., 10000)
S = 10.; R = 28.; B = 8./3.
trajectory = scipy.integrate.odeint(Lorenz,w0,timepoints,args=(S,R,B))

# trajectory is a scipy array of shape 10000 x 3
```

scipy provides functionality for integration, optimization, fitting, root-finding, special functions, FFTs, etc.

pylab (a.k.a. matplotlib)

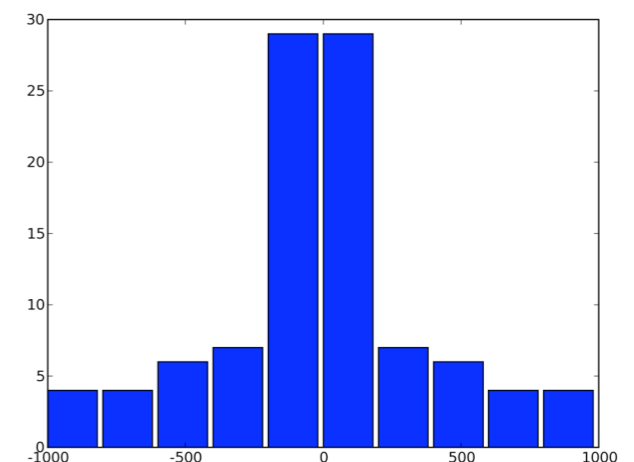
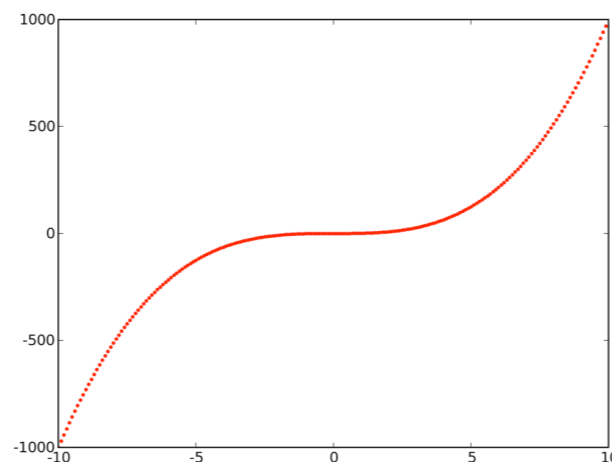
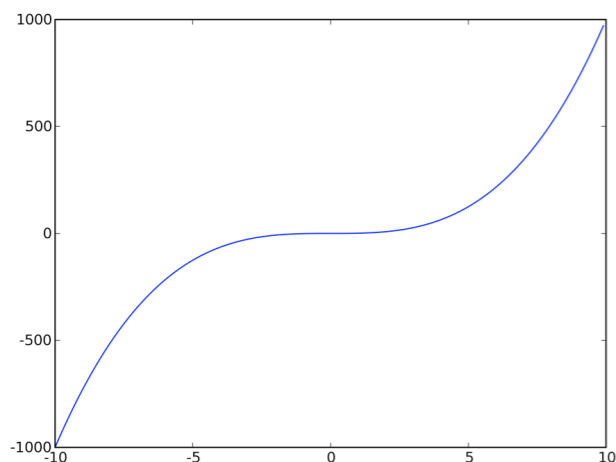
- (mostly) 2D plotting package based largely on Matlab plotting syntax

```
import pylab, scipy          # pylab can plot Python lists or scipy arrays

xvals = scipy.linspace(-10., 10., 100)    # equally spaced points in x
yvals = xvals**3                      # y = x**3 (x to power 3)
pylab.plot(xvals, yvals)              # plot yvals vs. xvals
pylab.show()                          # display plot on screen
pylab.plot(xvals, yvals, 'r.')         # plot with red dots
pylab.hist(yvals)                      # histogram of yvals

# control of labels, legends, tickmarks, line width, etc.

pylab.xlabel("This is the x coordinate")
pylab.ylabel("This is not the x coordinate")
```



Some third party libraries for scientific computing

- Numerics
 - scipy* (and numpy*)
 - ▶ **when possible, use “array syntax” for efficient (compiled) computation and compact expression**
- Interpreters / integrated computing environments
 - ipython*, IDLE, sage (incl. symbolic math)
- Graphics and visualization
 - pylab*, PIL* (Python Imaging Library), VPython*, pygnuplot, VTK/Mayavi
- Application-specific packages
 - NetworkX*, Biopython, SloppyCell

* used in course