ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Numpy for Scientists and Engineers

Enthought, Inc.
www.enthought.com

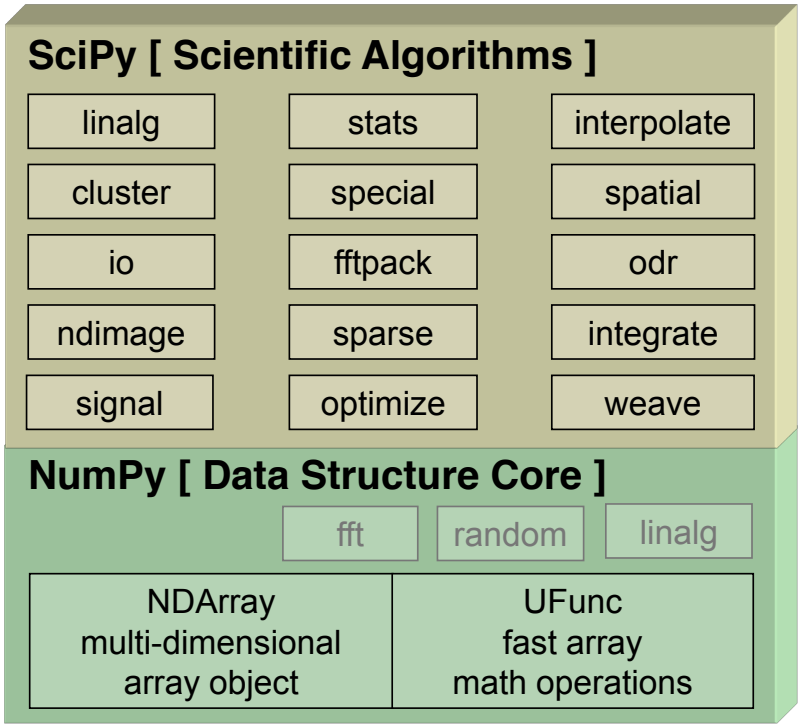
Enthought, Inc.
515 Congress Avenue
Suite 2100
Austin, TX 78701

www.enthought.com

8bcab43

Q2-2015

NumPy and SciPy



NumPy

- Website: http://numpy.scipy.org/

- Offers Matlab-ish capabilities within Python
- NumPy replaces Numeric and Numarray
- Initially developed by Travis Oliphant
- 225 “committers” to the project (github.com)
- NumPy 1.0 released October, 2006
- NumPy 1.8.1 released March, 2014
- ~200K downloads/month from PyPI

This does not count Linux distributions, MacOS ships with numpy, Enthought Canopy and other distributions, ...

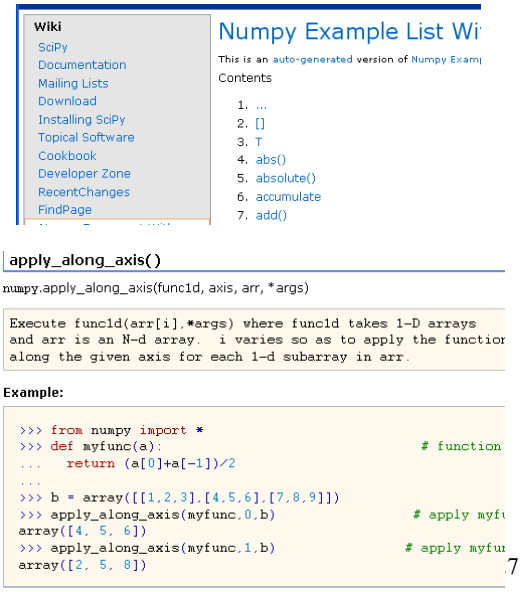
Helpful Sites

SCIPY DOCUMENTATION PAGE

http://docs.scipy.org/doc

NUMPY EXAMPLES

http://www.scipy.org/Numpy_Example_List_With_Doc



Getting Started

IMPORT NUMPY

```
In [1]: from numpy import *
In [2]: __version__
Out[2]: 1.8.1
```

or

```
In [1]: from numpy import \
   array, ...
```

Often at the command line, it is handy to import everything from NumPy into the command shell.

However, if you are writing scripts, it is easier for others to read and debug in the future if you use explicit imports.

USING IPYTHON -PYLAB

```
C:\> ipython --pylab
In [1]: array([1,2,3])
Out[1]: array([1, 2, 3])
```

IPython has a 'pylab' mode where it imports all of NumPy, Matplotlib, and SciPy into the namespace for you as a convenience. It also enables threading for showing plots.

While IPython is used for all the demos, '>>>' is used on future slides instead of '`In [1]:`' to save space.

Array Operations

SIMPLE ARRAY MATH

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

NumPy defines these constants:
pi = 3.14159265359
e = 2.71828182846

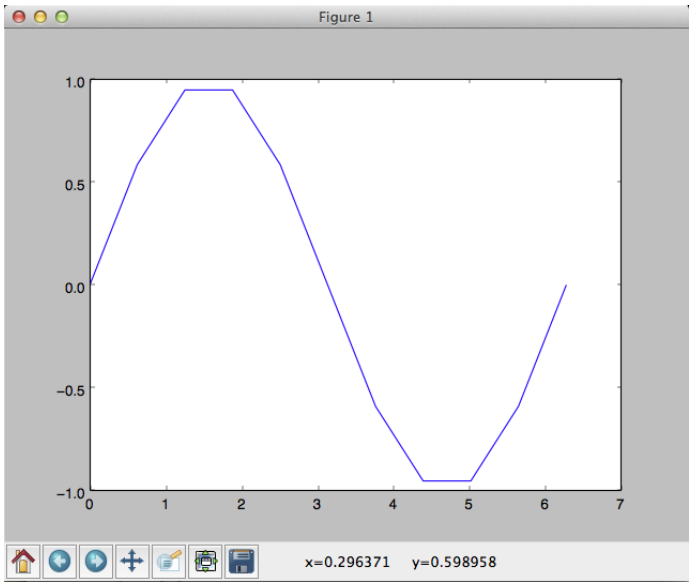
MATH FUNCTIONS

```
# create array from 0 to 10
>>> x = arange(11.)

# multiply entire array by
# scalar value
>>> c = (2*pi)/10.
>>> c
0.62831853071795862
>>> c*x
array([ 0.,0.628,…,6.283])

# in-place operations
>>> x *= c
>>> x
array([ 0.,0.628,…,6.283])

# apply functions to array
>>> y = sin(x)
```


Plotting Arrays

MATPLOTLIB

```
>>> plot(x,y)
```



Matplotlib Basics (an interlude)

http://matplotlib.org/

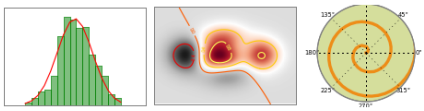
matplotlib

home | search | examples | gallery | docs » modules | index

intro

matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits.

matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code. For a sampling, see the screenshots, thumbnail gallery, and examples directory



For example, using "ipython -pylab" to provide an interactive environment, to generate 10,000 gaussian random numbers and plot a histogram with 100 bins, you simply need to type

```
x = randn(10000)
hist(x, 100)
```

For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users. The pylab mode provides all of the pyplot plotting functions listed below, as well as non-plotting functions from numpy and matplotlib.mlab.

plotting commands

Function	Description
acorr	plot the autocorrelation function

News

Please donate to support matplotlib development.

matplotlib 1.0.1 is available for download. See what's new and tips on installing

Sandro Tosi has a new book Matplotlib for python developers also at amazon.

Build websites like matplotlib's, with sphinx and extensions for mpl plots, math, inheritance diagrams -- try the sampledoc tutorial.

Videos

Watch the SciPy 2009 intro and advanced matplotlib tutorials

Watch a talk about matplotlib presented at NIPS 08 Workshop *MLOSS* and one presented at ChiPy.

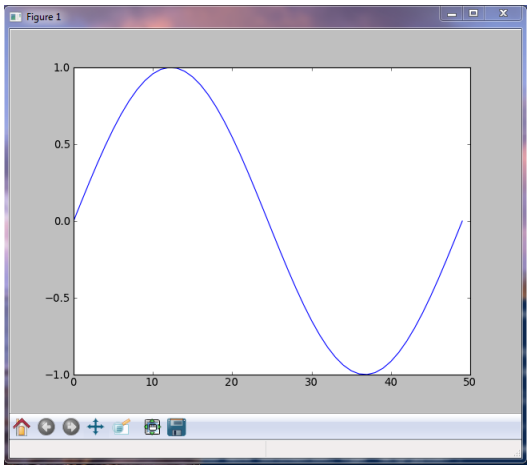
Toolkits

There are several matplotlib addon toolkits, including the projection and mapping toolkit

Line Plots

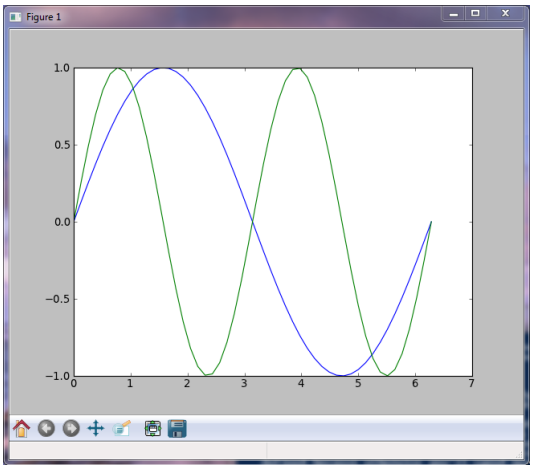
PLOT AGAINST INDICES

```
>>> x = linspace(0,2*pi,50)
>>> plot(sin(x))
```

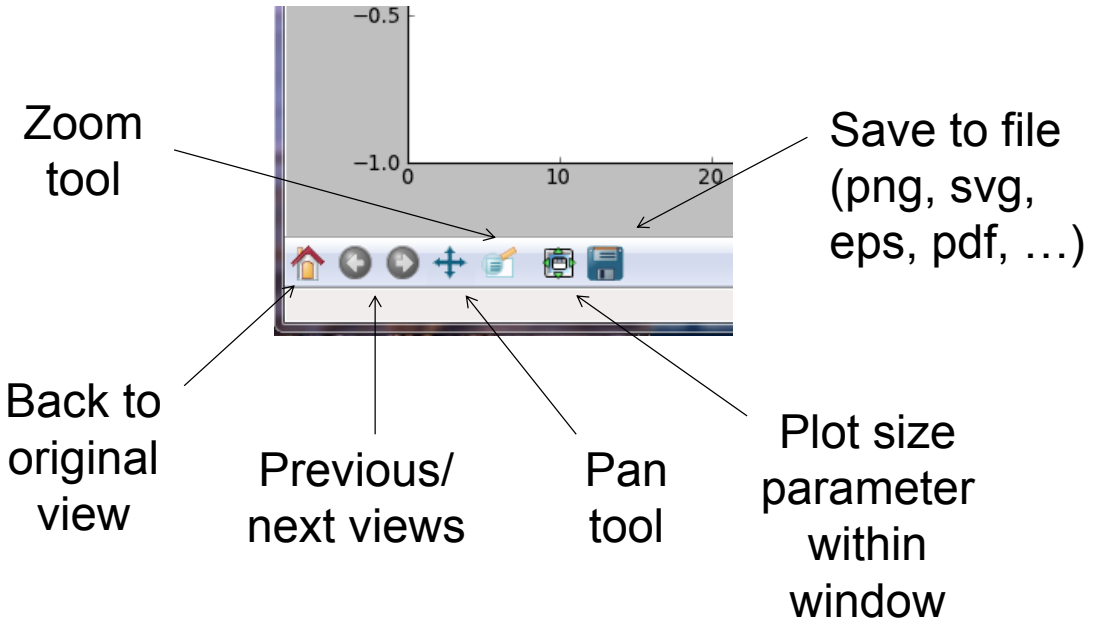


MULTIPLE DATA SETS

```
>>> plot(x, sin(x),
...      x, sin(2*x))
```



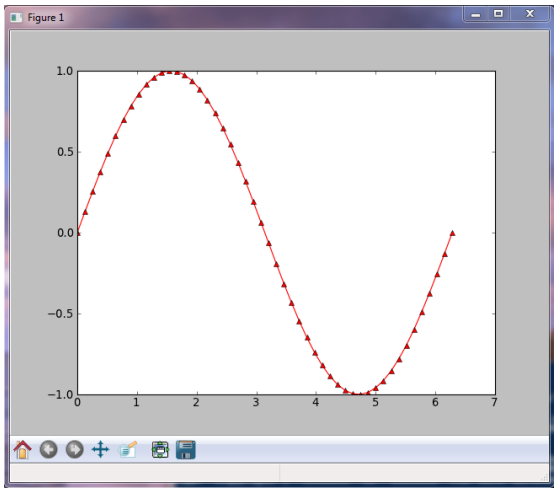
Matplotlib Menu Bar



Line Plots

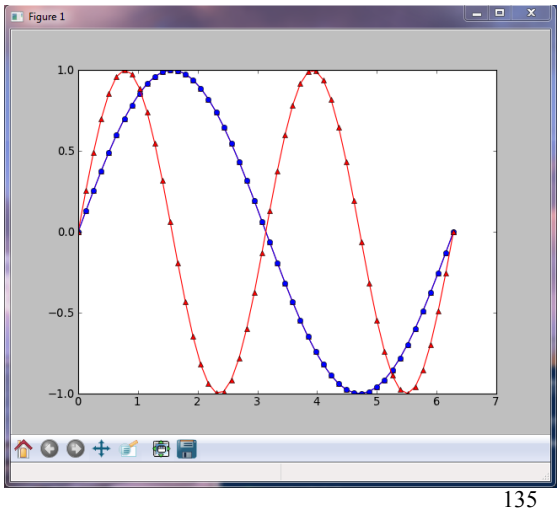
LINE FORMATTING

```
# red, dot-dash, triangles
>>> plot(x, sin(x), 'r-^')
```



MULTIPLE PLOT GROUPS

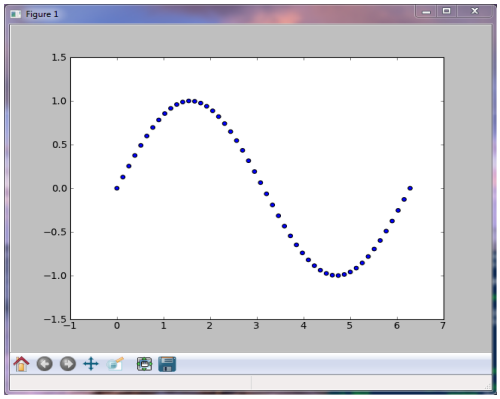
```
>>> plot(x, sin(x), 'b-o',
...      x, sin(2*x), 'r-^')
```



Scatter Plots

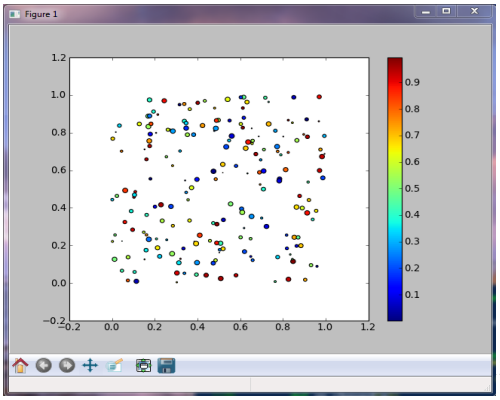
SIMPLE SCATTER PLOT

```
>>> x = linspace(0,2*pi,50)
>>> y = sin(x)
>>> scatter(x, y)
```



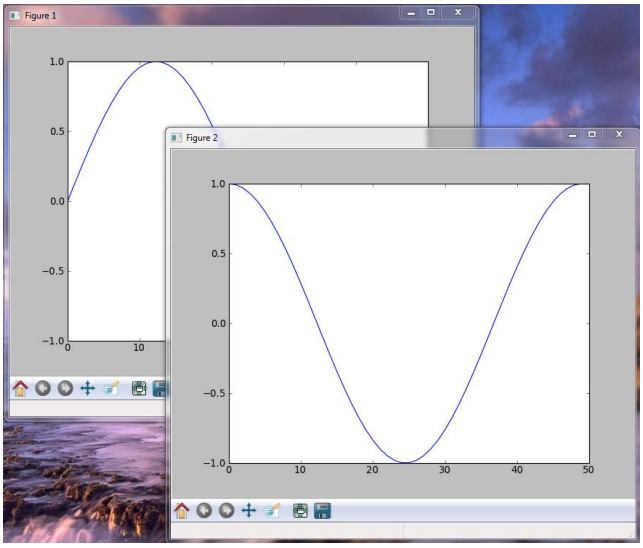
COLORMAPPED SCATTER

```
# marker size/color set with data
>>> x = rand(200)
>>> y = rand(200)
>>> size = rand(200)*30
>>> color = rand(200)
>>> scatter(x, y, size, color)
>>> colorbar()
```



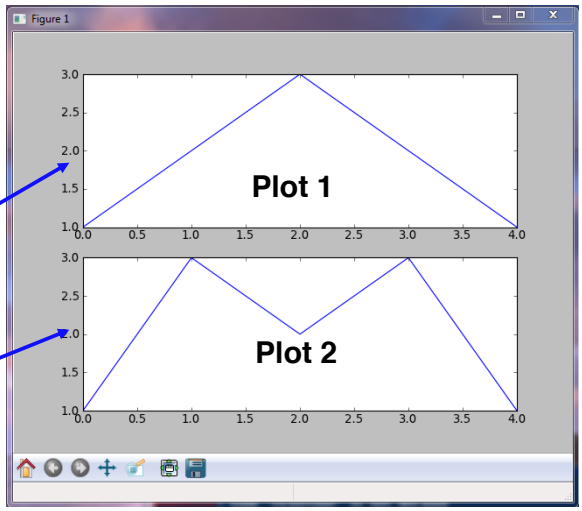
Multiple Figures

```
>>> t = linspace(0,2*pi,50)
>>> x = sin(t)
>>> y = cos(t)

# Now create a figure
>>> figure()
>>> plot(x)

# Now create a new figure.
>>> figure()
>>> plot(y)
```



Multiple Plots Using `subplot`

```
>>> x = array([1,2,3,2,1])
>>> y = array([1,3,2,3,1])

# To divide the plotting area
#                columns
#                   |
>>> subplot(2, 1, 1)
>>> plot(x)  |      |
#          rows   active plot


# Now activate a new plot
# area.
>>> subplot(2, 1, 2)
>>> plot(y)
```

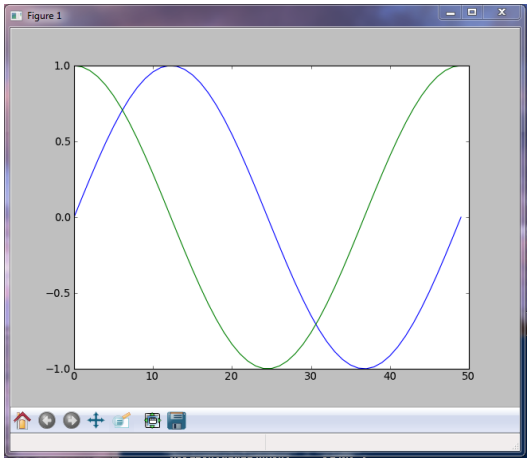


If this is used in a python script, a call to the function show() is required.

Adding Lines to a Plot

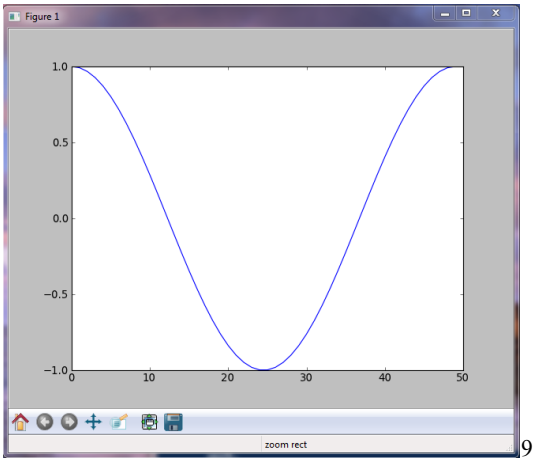
MULTIPLE PLOTS

```
# By default, previous lines
# are "held" on a plot.
>>> plot(sin(x))
>>> plot(cos(x))
```



ERASING OLD PLOTS

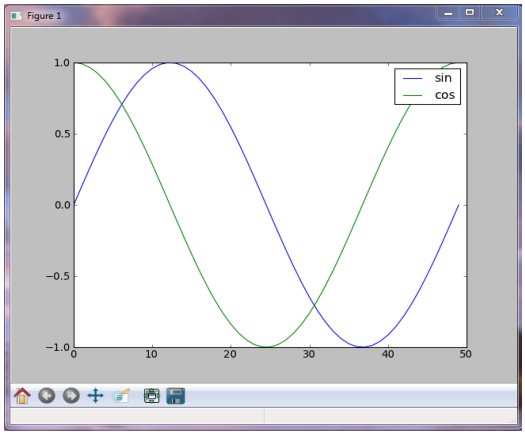
```
# Set hold(False) to erase
# old lines
>>> plot(sin(x))
>>> hold(False)
>>> plot(cos(x))
```



Legend

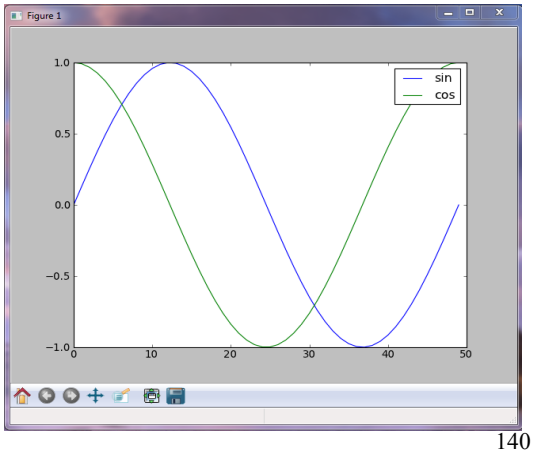
LEGEND LABELS WITH PLOT

```
# Add labels in plot command.
>>> plot(sin(x), label='sin')
>>> plot(cos(x), label='cos')
>>> legend()
```



LABELING WITH LEGEND

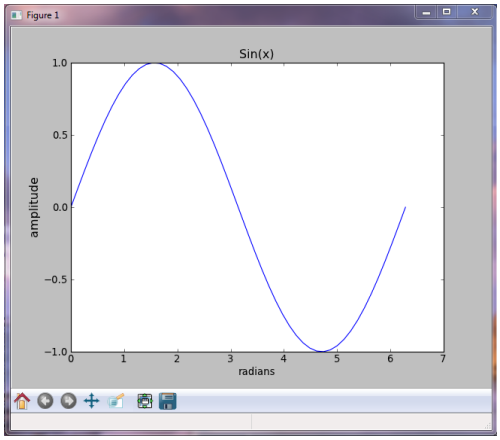
```
# Or as a list in legend().
>>> plot(sin(x))
>>> plot(cos(x))
>>> legend(['sin', 'cos'])
```



Titles and Grid

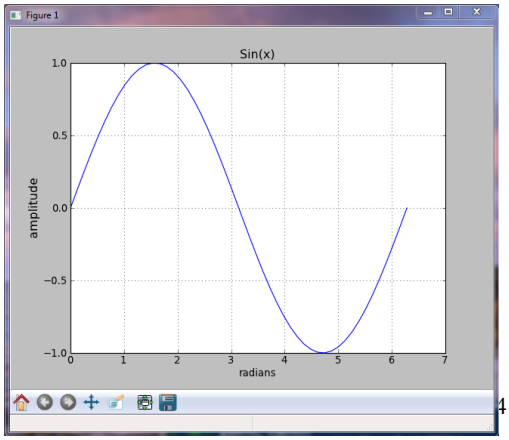
TITLES AND AXIS LABELS

```
>>> plot(x, sin(x))
>>> xlabel('radians')
# Keywords set text properties.
>>> ylabel('amplitude',
...         fontsize='large')
>>> title('Sin(x)')
```



PLOT GRID

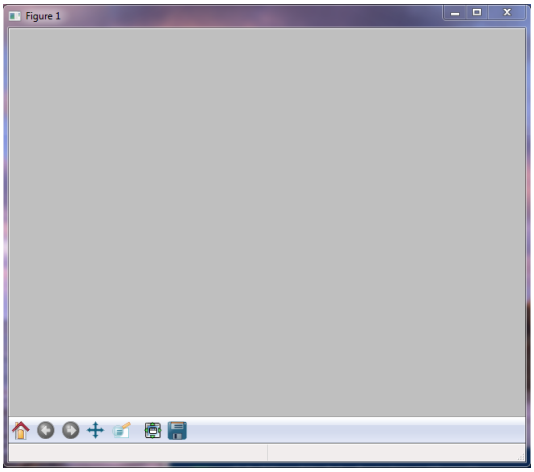
```
# Display gridlines in plot
>>> grid()
```



Clearing and Closing Plots

CLEARING A FIGURE

```
>>> plot(x, sin(x))
# clf will clear the current
# plot (figure).
>>> clf()
```

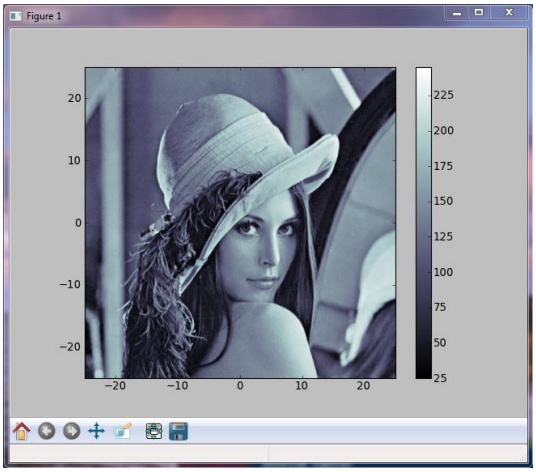


CLOSING PLOT WINDOWS

```
# close() will close the
# currently active plot window.
>>> close()

# close('all') closes all the
# plot windows.
>>> close('all')
```

Image Display

```
# Get the Lena image from scipy.
>>> from scipy.misc import lena
>>> img = lena()

# Display image with the jet
# colormap, and setting
# x and y extents of the plot.
>>> imshow(img,
...         extent=[-25,25,-25,25],
...         cmap = cm.bone)

# Add a colorbar to the display.
>>> colorbar()
```



Plotting from Scripts

INTERACTIVE MODE

```
# In IPython, plots show up
# as soon as a plot command
# is called.
>>> figure()
>>> plot(sin(x))
>>> figure()
>>> plot(cos(x))
```

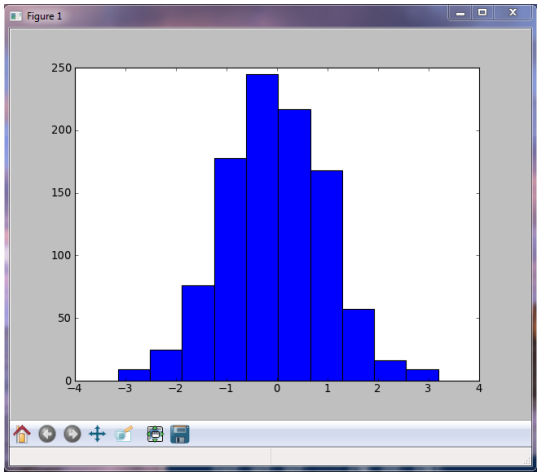
NON-INTERACTIVE MODE

```
# script.py
# In a script, you must call
# the show() command to display
# plots. Call it at the end of
# all your plot commands for
# best performance.
figure()
plot(sin(x))
figure()
plot(cos(x))

# Plots will not appear until
# this command is issued.
show()
```

Histograms

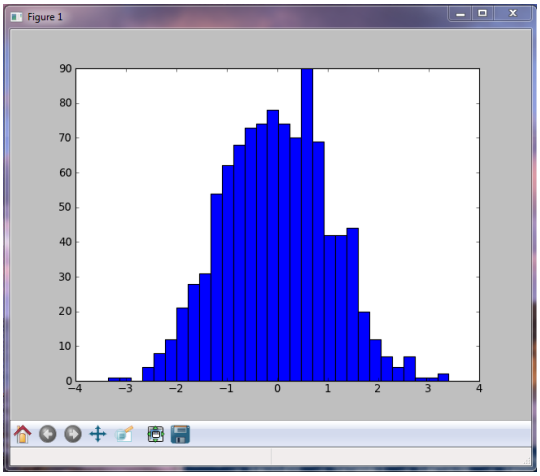
HISTOGRAM

```
# plot histogram
# defaults to 10 bins
>>> hist(randn(1000))
```



HISTOGRAM 2

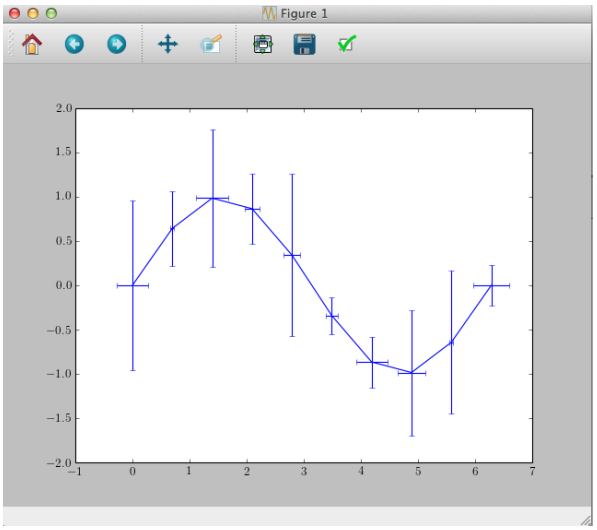
```
# change the number of bins
>>> hist(randn(1000), 30)
```



Plots with error bars

ERRORBAR

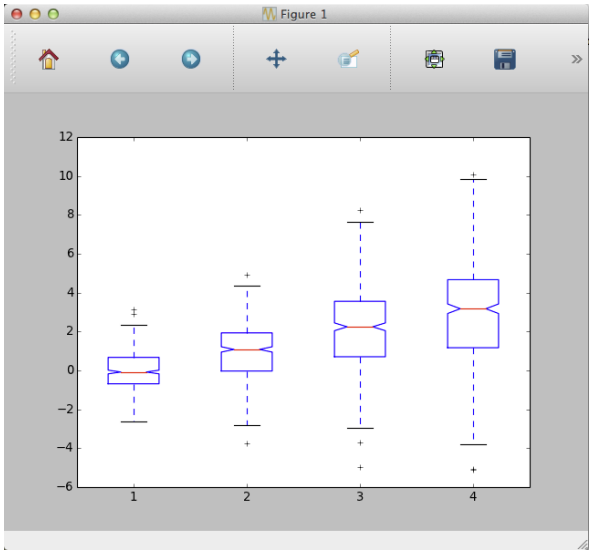
```
# Assume points are known
# with errors in both axis
>>> x = linspace(0,2*pi,10)
>>> y = sin(x)
>>> yerr = rand(10)
>>> xerr = rand(10)/3
>>> errorbar(x, y, yerr, xerr)
```



Plots with error bars II

BOXPLOT

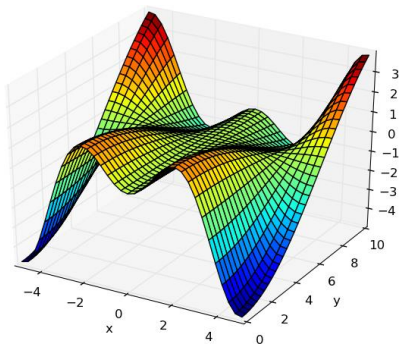
```
# Assume 4 experiments have measured some
# quantities
# Data creation
>>> from numpy.random import normal
>>> ex = [normal(i, 1+i/2, size=(500,))
          for i in np.arange(4.)]


# Plot creation
>>> positions = np.arange(len(ex)))+1
>>> boxplot(ex, sym='k+', notch=True,
            positions=positions)


# Interpretation
# Red line = median
# Notch = median @95% CL
# Box = [25%-75%]
# Whiskers = 1.5 * box_extent
```



3D Plots with Matplotlib

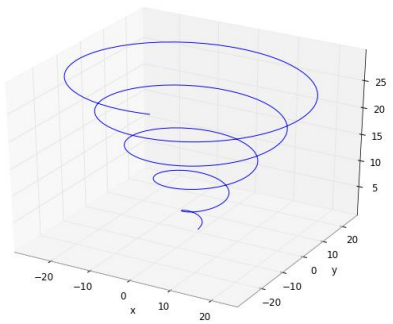
SURFACE PLOT

```
>>> from mpl_toolkits.mplot3d import
Axes3D
>>> x, y = mgrid[-5:5:35j, 0:10:35j]
>>> z = x*sin(x)*cos(0.25*y)
>>> fig = figure()
>>> ax = fig.gca(projection='3d')
>>> ax.plot_surface(x, y, z,
...      rstride=1, cstride=1,
...      cmap=cm.jet)
>>> xlabel('x'); ylabel('y')
```



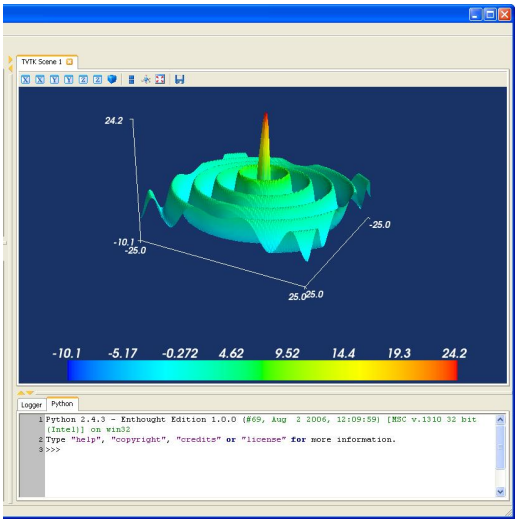
PARAMETRIC CURVE

```
>>> from mpl_toolkits.mplot3d import
Axes3D
>>> t = linspace(0, 30, 1000)
>>> x, y, z = [t*cos(t), t*sin(t), t]
>>> fig = figure()
>>> ax = fig.gca(projection='3d')
>>> ax.plot(x, y, z)
>>> xlabel('x')
>>> ylabel('y')
```



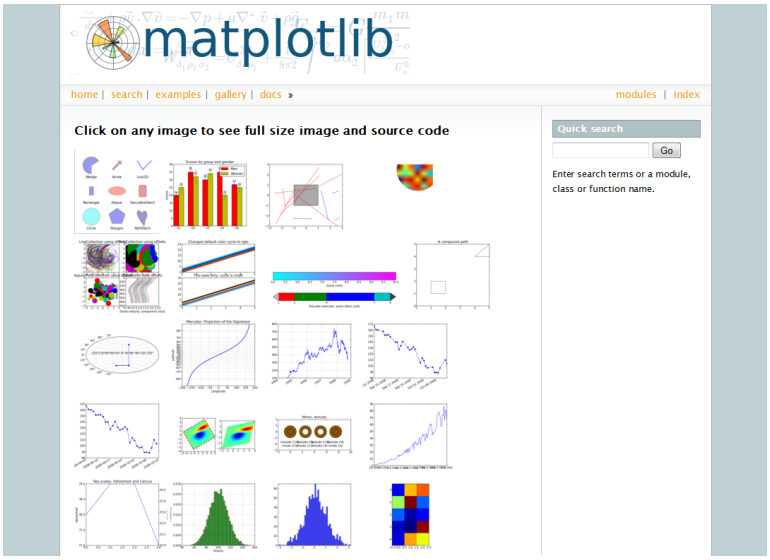
Surface Plots with `mlab`

```
# Create 2D array where values
# are radial distance from
# the center of array.
>>> from numpy import mgrid
>>> from scipy import special
>>> x,y = mgrid[-25:25:100j,
...             -25:25:100j]
>>> r = sqrt(x**2+y**2)
# Calculate Bessel function of
# each point in array and scale.
>>> s = special.j0(r)*25

# Display surface plot.
>>> from mayavi import mlab
>>> mlab.surf(x,y,s)
>>> mlab.scalarbar()
>>> mlab.axes()
```



More Details

- Simple examples with increasing difficulty: http://matplotlib.org/examples/index.html
- Gallery (huge): http://matplotlib.org/gallery.html



Continuing NumPy…

Introducing NumPy Arrays

SIMPLE ARRAY CREATION

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

CHECKING THE TYPE

```
>>> type(a)
numpy.ndarray
```

NUMERIC 'TYPE' OF ELEMENTS

```
>>> a.dtype
dtype('int32')
```

BYTES PER ELEMENT

```
>>> a.itemsize
4
```

ARRAY SHAPE

```
# Shape returns a tuple
# listing the length of the
# array along each dimension.
>>> a.shape
(4,)
>>> shape(a)
(4,)
```

ARRAY SIZE

```
# Size reports the entire
# number of elements in an
# array.
>>> a.size
4
>>> size(a)
4
```

Introducing NumPy Arrays

ENTHOUGHT

BYTES OF MEMORY USED

```
# Return the number of bytes
# used by the data portion of
# the array.
 >>> a.nbytes
16
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
1
```


Setting Array Elements

ENTHOUGHT

ARRAY INDEXING

```
>>> a[0]
0
>>> a[0] = 10
>>> a
array([10, 1, 2, 3])
```

FILL

```
# set all values in an array
>>> a.fill(0)
>>> a
array([0, 0, 0, 0])

# this also works, but may
# be slower
>>> a[:] = 1
>>> a
array([1, 1, 1, 1])
```

BEWARE OF TYPE COERCION

```
>>> a.dtype
dtype('int32')

# assigning a float into
# an int32 array truncates
# the decimal part
>>> a[0] = 10.6
>>> a
array([10, 1, 2, 3])

# fill has the same behavior
>>> a.fill(-4.8)
>>> a
array([-4, -4, -4, -4])
```

Slicing

`var[lower:upper:step]`

Extracts a portion of a sequence by specifying a lower and upper bound.
The lower-bound element is included, but the upper-bound element is **not** included.
Mathematically: [lower, upper). The step value specifies the stride between elements.

SLICING ARRAYS

```
# indices:    0  1  2  3  4
>>> a = array([10,11,12,13,14])
# [10,11,12,13,14]
>>> a[1:3]
array([11, 12])

# negative indices work also
>>> a[1:-2]
array([11, 12])
>>> a[-4:3]
array([11, 12])
```

OMITTING INDICES

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list

# grab first three elements
>>> a[:3]
array([10, 11, 12])
# grab last two elements
>>> a[-2:]
array([13, 14])
# every other element
>>> a[::2]
array([10, 12, 14])
```


Multi-Dimensional Arrays

MULTI-DIMENSIONAL ARRAYS

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

SHAPE = (ROWS,COLUMNS)

```
>>> a.shape
(2, 4)
```

ELEMENT COUNT

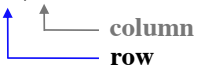
```
>>> a.size
8
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
2
```

GET/SET ELEMENTS

```
>>> a[1,3]
13
```




```
>>> a[1,3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```

ADDRESS SECOND (ONETH) ROW USING SINGLE INDEX

```
>>> a[1]
array([10, 11, 12, -1])
```

Arrays from/to ASCII files

BASIC PATTERN

```
# Read data into a list of lists,
# and THEN convert to an array.
file = open('myfile.txt')

# Create a list for all the data.
data = []

for line in file:
  # Read each row of data into a
  # list of floats.
  fields = line.split()
  row_data = [float(x) for x
                          in fields]
  # And add this row to the
  # entire data set.
  data.append(row_data)

# Finally, convert the "list of
# lists" into a 2D array.
data = array(data)
file.close()
```

ARRAYS FROM/TO TXT FILES

Data.txt

```
-- BEGINNING OF THE FILE
% Day,  Month,  Year, Skip, Avg Power
01, 01, 2000, x876, 13 % crazy day!
% we don't have Jan 03rd
04, 01, 2000, xfed, 55
```

```
# loadtxt() automatically generates
# an array from the txt file
arr = loadtxt('Data.txt', skiprows=1,
   dtype=int, delimiter=",",
   usecols = (0,1,2,4),
   comments = "%")

# Save an array into a txt file
savetxt('filename', arr)
```


Arrays to/from Files

OTHER FILE FORMATS

Many file formats are supported in various packages:

File format	Package name(s)	Functions
txt	numpy	loadtxt, savetxt, genfromtxt, fromfile, tofile
csv	csv	reader, writer
Matlab	scipy.io	loadmat, savemat
hdf	pytables, h5py	
NetCDF	netCDF4, scipy.io.netcdf	netCDF4.Dataset, scipy.io.netcdf.netcdf_file

This includes many industry specific formats:

File format	Package name	Comments
wav	scipy.io.wavfile	Audio files
LAS/SEG-Y	Scipy cookbook, Obspy	Data files in Geophysics
jpeg, png, …	PIL, scipy.misc.pilutil	Common image formats
FITS	pyfits, astropy.io.fits	Image files in Astronomy

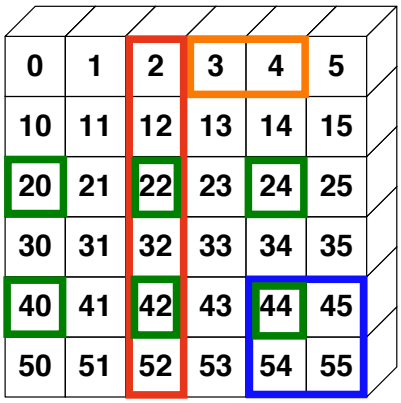
Array Slicing

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

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

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])
```

STRIDES ARE ALSO POSSIBLE

```
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



Slices Are References

Slices are references to memory in the original array.

Changing values in a slice also changes the original array.

```
>>> a = array((0,1,2,3,4))

# create a slice containing only the
# last element of a
>>> b = a[2:4]
>>> b
array([2, 3])
>>> b[0] = 10

# changing b changed a!
>>> a
array([ 0,  1, 10, 3, 4])
```

Fancy Indexing

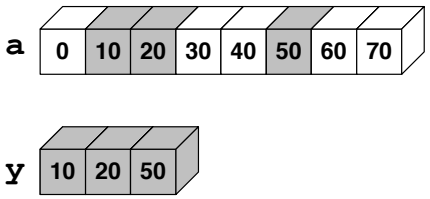
ENTHOUGHT

INDEXING BY POSITION

```
>>> a = arange(0,80,10)


# fancy indexing
>>> indices = [1, 2, -3]
>>> y = a[indices]
>>> print(y)
[10 20 50]
```

INDEXING WITH BOOLEANS

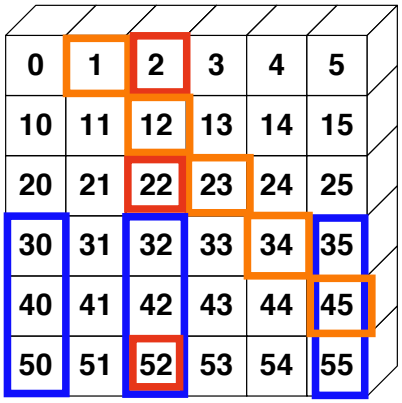
```
# manual creation of masks
>>> mask = array([0,1,1,0,0,1,0,0],
...              dtype=bool)

# conditional creation of masks
>>> mask2 = a < 30

# fancy indexing
>>> y = a[mask]
>>> print(y)
[10 20 50]
```



Fancy Indexing in 2-D

ENTHOUGHT

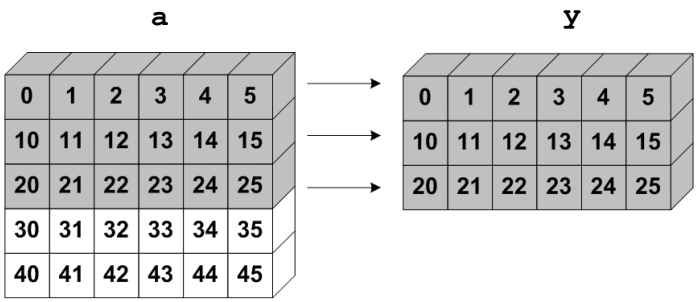
```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



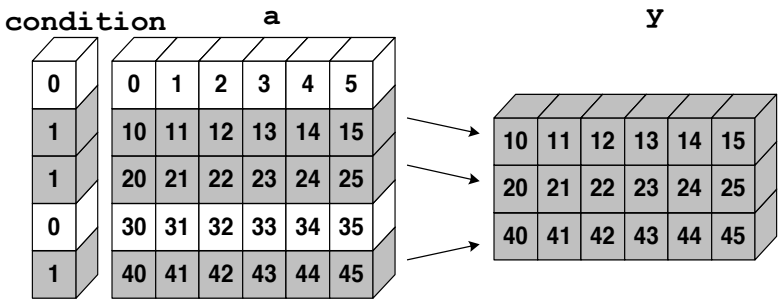
Unlike slicing, fancy indexing creates copies instead of a view into original array.

“Incomplete” Indexing

```
>>> y = a[:3]
```



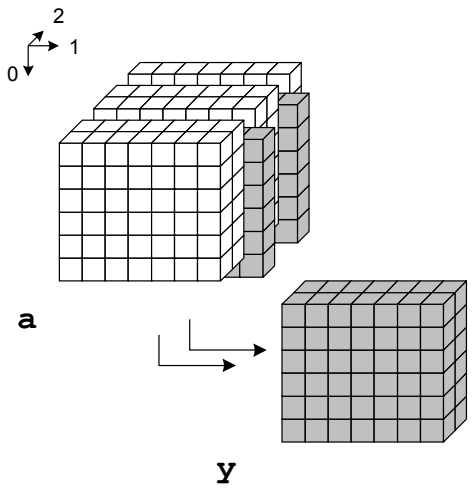

```
>>> y = a[condition]
```



3D Example

MULTIDIMENSIONAL

```
# retrieve two slices from a
# 3D cube via indexing
>>> y = a[:,:,[2,-2]]
```



Where

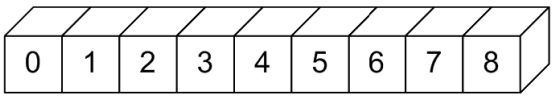
1 DIMENSION

```
# find the indices in array
# where expression is True
>>> a = array([0, 12, 5, 20])
>>> a > 10
array([False, True, False,
       True], dtype=bool)

# Note: it returns a tuple!
>>> where(a > 10)
(array([1, 3]),)
```

n DIMENSIONS

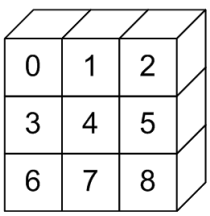
```
# In general, the tuple
# returned is the index of the
# element satisfying the
# condition in each dimension.
>>> a = array([[0, 12, 5, 20],
               [1, 2, 11, 15]])
>>> loc = where(a > 10)
>>> loc
(array([0, 0, 1, 1]),
array([1, 3, 2, 3]))

# Result can be used in
# various ways:
>>> a[loc]
array([12, 20, 11, 15])
```


Array Data Structure

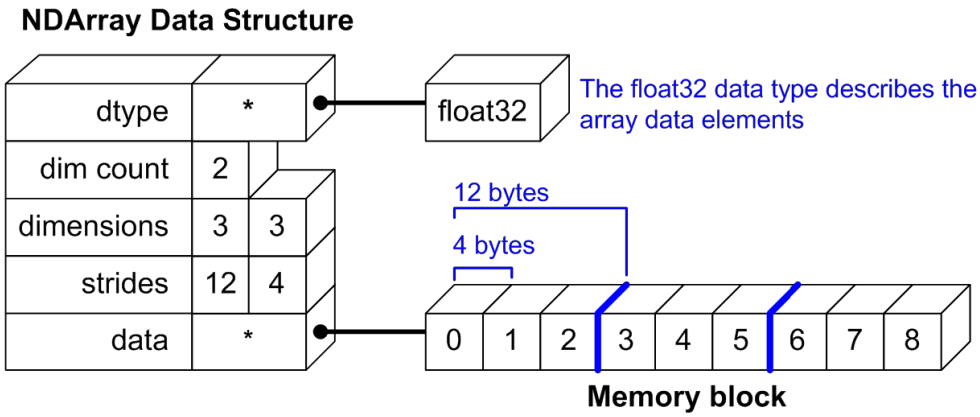


Memory block

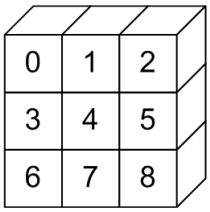
Python View:



Array Data Structure



Python View:



"Flattening" Arrays

a.flatten()

`a.flatten()` converts a multi-dimensional array into a 1-D array. The new array is a *copy* of the original data.

```
# Create a 2D array
>>> a = array([[0,1],
               [2,3]])

# Flatten out elements to 1D
>>> b = a.flatten()
>>> b
array([0,1,2,3])

# Changing b does not change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[0, 1],
       [2, 3]])
```

no change

a.flat

`a.flat` is an *attribute* that returns an iterator object that accesses the data in the multi-dimensional array data as a 1-D array. It *references* the original memory.

```
>>> a.flat
<numpy.flatiter obj...>
>>> a.flat[:]
array([0,1,2,3])

>>> b = a.flat
>>> b[0] = 10
>>> a
array([[10, 1],
       [ 2, 3]])
```

changed!

"(Un)raveling" Arrays

a.ravel()

`a.ravel()` is the same as `a.flatten()`, but returns a *reference (or view)* of the array if possible (i.e., the memory is contiguous). Otherwise the new array copies the data.

```
# create a 2-D array
>>> a = array([[0,1],
               [2,3]])

# flatten out elements to 1-D
>>> b = a.ravel()
>>> b
array([0,1,2,3])

# changing b does change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[10, 1],
       [ 2, 3]])
```

changed!

a.ravel() MAKES A COPY

```
# create a 2-D array
>>> a = array([[0,1],
               [2,3]])

# transpose array so memory
# layout is no longer contiguous
>>> aa = a.transpose()
>>> aa
array([[0, 2],
       [1, 3]])
# ravel creates a copy of data
>>> b = aa.ravel()
array([0,2,1,3])
# changing b doesn't change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[0, 1],
       [2, 3]])
```


Reshaping Arrays

SHAPE

```
>>> a = arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.shape
(6,)


# reshape array in-place to
# 2x3
>>> a.shape = (2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
```

RESHAPE

```
# return a new array with a
# different shape
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])


# reshape cannot change the
# number of elements in an
# array
>>> a.reshape(4,2)
ValueError: total size of new
array must be unchanged
```

Transpose

TRANSPOSE

```
>>> a = array([[0,1,2],
...            [3,4,5]])
>>> a.shape
(2,3)
# Transpose swaps the order
# of axes. For 2-D this
# swaps rows and columns.
>>> a.transpose()
array([[0, 3],
       [1, 4],
       [2, 5]])

# The .T attribute is
# equivalent to transpose().
>>> a.T
array([[0, 3],
       [1, 4],
       [2, 5]])
```

TRANSPOSE RETURNS VIEWS

```
>>> b = a.T

# Changes to b alter a.
>>> b[0,1] = 30
>>> a
array([[ 0,  1,  2],
       [30,  4,  5]])
```

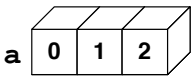
TRANSPOSE AND STRIDES

```
# Transpose does not move
# values around in memory. It
# only changes the order of
# "strides" in the array
>>> a.strides
(12, 4)

>>> a.T.strides
(4, 12)
```


Indexing with newaxis

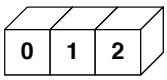
`newaxis` is a special index that inserts a new axis in the array at the specified location.

Each **`newaxis`** increases the array's dimensionality by 1.



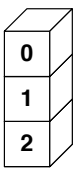
1 X 3

```
>>> shape(a)
(3,)
>>> y = a[newaxis,:]
>>> shape(y)
(1, 3)
```



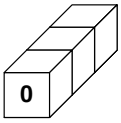
3 X 1

```
>>> y = a[:,newaxis]
>>> shape(y)
(3, 1)
```



1 X 1 X 3

```
> y = a[newaxis, newaxis, :]
> shape(y)
(1, 1, 3)
```



Squeeze

ENTHOUGHT

SQUEEZE

```
>>> a = array([[1,2,3],
...            [4,5,6]])
>>> a.shape
(2,3)

# insert an "extra" dimension
>>> a.shape = (2,1,3)
>>> a
array([[[0, 1, 2]],
       [[3, 4, 5]]])

# squeeze removes any
# dimension with length==1
>>> a = a.squeeze()
>>> a.shape
(2,3)
```


Diagonals

ENTHOUGHT

DIAGONAL

```
>>> a = array([[11,21,31],
...            [12,22,32],
...            [13,23,33]])


# Extract the diagonal from
# an array.
>>> a.diagonal()
array([11, 22, 33])

# Use offset to move off the
# main diagonal (offset can
# be negative).
>>> a.diagonal(offset=1)
array([21, 32])
```

DIAGONALS WITH INDEXING

```
# "Fancy" indexing also works.
>>> i = [0,1,2]
>>> a[i, i]
array([11, 22, 33])

# Indexing can also be used
# to set diagonal values…
>>> a[i, i] = 2
>>> i2 = array([0,1])
# upper diagonal
>>> a[i2, i2+1] = 1
# lower diagonal
>>> a[i2+1, i2]= -1
>>> a
array([[ 2,  1, 31],
       [-1,  2,  1],
       [13, -1,  2]])
```

Complex Numbers

COMPLEX ARRAY ATTRIBUTES

```
>>> a = array([1+1j, 2, 3, 4])
array([1.+1.j, 2.+0.j, 3.+0.j,
       4.+0.j])
>>> a.dtype
dtype('complex128')

# real and imaginary parts
>>> a.real
array([ 1.,  2.,  3.,  4.])
>>> a.imag
array([ 1.,  0.,  0.,  0.])

# set imaginary part to a
# different set of values
>>> a.imag = (1,2,3,4)
>>> a
array([1.+1.j, 2.+2.j, 3.+3.j,
       4.+4.j])
```

CONJUGATION

```
>>> a.conj()
array([1.-1.j, 2.-2.j, 3.-3.j,
       4.-4.j])
```

FLOAT (AND OTHER) ARRAYS

```
>>> a = array([0., 1, 2, 3])

# .real and .imag attributes
# are available
>>> a.real
array([ 0., 1., 2., 3.])
>>> a.imag
array([ 0., 0., 0., 0.])

# but .imag is read-only
>>> a.imag = (1,2,3,4)
TypeError: array does not
have imaginary part to set
```


Array Constructor Examples

FLOATING POINT ARRAYS

```
# Default to double precision
>>> a = array([0,1.0,2,3])
>>> a.dtype
dtype('float64')
>>> a.nbytes
32
```

REDUCING PRECISION

```
>>> a = array([0,1.,2,3],
...           dtype=float32)
>>> a.dtype
dtype('float32')
>>> a.nbytes
16
```

UNSIGNED INTEGER BYTE

```
>>> a = array([0,1,2,3],
...           dtype=uint8)
>>> a.dtype
dtype('uint8')
>>> a.nbytes
4
```

ARRAY FROM BINARY DATA

```
# frombuffer or fromfile
# to create an array from
# binary data.
>>> a = frombuffer('foo',
...           dtype=uint8)
>>> a
array([102, 111, 111])
# Reverse operation
>>> a.tofile('foo.dat')
```

Specifying DTypes

DEFAULT (BY INSPECTION)

```
# float -> np.float64
>>> a = array([0,1.0,2,3])
>>> a.dtype
dtype('float64')

# int -> np.int64
>>> b = array([0,1,2,3])
>>> b.dtype
dtype('int64')
```

PYTHON DATA TYPES

```
# float -> np.float64
>>> c = array([0,1,2,3],
              dtype=float)
>>> c.dtype
dtype('float64')
```

NUMPY DATA TYPES

```
>>> from numpy import uint8
>>> d = array([0,1,2,3],
...           dtype=uint8)
>>> d.dtype
dtype('uint8')
```

STRING SPECIFICATION

```
# Big-Endian float, 8 bytes
>>> e = array([0,1,2,3],
...           dtype=">f8")
>>> e.dtype
dtype('>f8')

# Strings of length 8
>>> f = array(["01234567"],
...           dtype="S8")
>>> f.dtype
dtype('S8')
```


NumPy dtypes

Basic Type	Available NumPy types	Code	Comments
Boolean	`bool`	`b`	Elements are 1 byte in size.
Integer	`int8, int16, int32, int64, int128, int`	`i`	int defaults to the size of long in C for the platform.
Unsigned Integer	`uint8, uint16, uint32, uint64, uint128, uint`	`u`	uint defaults to the size of unsigned long in C for the platform.
Float	`float16, float32, float64, float,longfloat`	`f`	float is always a double precision floating point value (64 bits). longfloat represents large precision floats. Its size is platform dependent.
Complex	`complex64, complex128, complex, longcomplex`	`c`	The real and imaginary elements of a complex64 are each represented by a single precision (32 bit) value for a total size of 64 bits.
Strings	`str, unicode`	`S` or `a, U`	For example, dtype='S4' would be used for an array of 4-character strings.
DateTime	`datetime64, timedelta64`	See section	Allow operations between dates and/or times. New in 1.7.
Object	`object`	`O`	Represent items in array as Python objects.
Records	`void`	`V`	Used for arbitrary data structures.

Type Casting

ASARRAY

```
>>> a = array([1.5, -3],
...           dtype=float32)
>>> a
array([ 1.5, -3.], dtype=float32)

# upcast
>>> asarray(a, dtype=float64)
array([ 1.5, -3. ])

# downcast
>>> asarray(a, dtype=uint8)
array([ 1, 253], dtype=uint8)

# asarray is efficient.
# It does not make a copy if the
# type is the same.
>>> b = asarray(a, dtype=float32)
>>> b[0] = 2.0
>>> a
array([ 2., -3.],dtype=float32)
```

ASTYPE

```
>>> a = array([1.5, -3],
...           dtype=float64)
>>> a.astype(float32)
array([ 1.5, -3.], dtype=float32)

>>> a.astype(uint8)
array([  1, 253],dtype=uint8)

# astype is safe.
# It always returns a copy of
# the array.
>>> b = a.astype(float64)
>>> b[0] = 2.0
>>> a
array([1.5, -3.])
```


Array Calculation Methods

SUM FUNCTION

```
>>> a = array([[1,2,3],
               [4,5,6]])

# sum() defaults to adding up
# all the values in an array.
>>> sum(a)
21

# supply the keyword axis to
# sum along the 0th axis
>>> sum(a, axis=0)
array([5, 7, 9])

# supply the keyword axis to
# sum along the last axis
>>> sum(a, axis=-1)
array([ 6, 15])
```

SUM ARRAY METHOD

```
# a.sum() defaults to adding
# up all values in an array.
>>> a.sum()
21

# supply an axis argument to
# sum along a specific axis
>>> a.sum(axis=0)
array([5, 7, 9])
```

PRODUCT

```
# product along columns
>>> a.prod(axis=0)
array([ 4, 10, 18])

# functional form
>>> prod(a, axis=0)
array([ 4, 10, 18])
```

Min/Max

MIN

```
>>> a = array([2.,3.,0.,1.])
>>> a.min(axis=0)
0.0
# Use NumPy's amin() instead
# of Python's built-in min()
# for speedy operations on
# multi-dimensional arrays.
>>> amin(a, axis=0)
0.0
```

MAX

```
>>> a = array([2.,3.,0.,1.])
>>> a.max(axis=0)
3.0


# functional form
>>> amax(a, axis=0)
3.0
```

ARGMIN

```
# Find index of minimum value.
>>> a.argmin(axis=0)
2
# functional form
>>> argmin(a, axis=0)
2
```

ARGMAX

```
# Find index of maximum value.
>>> a.argmax(axis=0)
1
# functional form
>>> argmax(a, axis=0)
1
```


Statistics Array Methods

MEAN

```
>>> a = array([[1,2,3],
               [4,5,6]])

# mean value of each column
>>> a.mean(axis=0)
array([ 2.5,  3.5,  4.5])
>>> mean(a, axis=0)
array([ 2.5,  3.5,  4.5])
>>> average(a, axis=0)
array([ 2.5,  3.5,  4.5])

# average can also calculate
# a weighted average
>>> average(a, weights=[1,2],
...         axis=0)
array([ 3., 4., 5.])
```

STANDARD DEV./VARIANCE

```
# Standard Deviation
>>> a.std(axis=0)
array([ 1.5,  1.5,  1.5])

# variance
>>> a.var(axis=0)
array([2.25, 2.25, 2.25])
>>> var(a, axis=0)
array([2.25, 2.25, 2.25])
```

Other Array Methods

CLIP

```
# Limit values to a range.

>>> a = array([[1,2,3],
               [4,5,6]])

# Set values < 3 equal to 3.
# Set values > 5 equal to 5.
>>> a.clip(3, 5)
array([[3, 3, 3],
       [4, 5, 5]])
```

PEAK TO PEAK

```
# Calculate max - min for
# array along columns
>>> a.ptp(axis=0)
array([3, 3, 3])
# max - min for entire array.
>>> a.ptp(axis=None)
5
```

ROUND

```
# Round values in an array.
# NumPy rounds to even, so
# 1.5 and 2.5 both round to 2.
>>> a = array([1.35, 2.5, 1.5])
>>> a.round()
array([ 1.,  2.,  2.])

# Round to first decimal place.
>>> a.round(decimals=1)
array([ 1.4,  2.5,  1.5])
```


Summary of (most) array attributes/methods (1/4)

	BASIC ATTRIBUTES
a.dtype	Numerical type of array elements: float 32, uint8, etc.
a.shape	Shape of array (m, n, o, …)
a.size	Number of elements in entire array
a.itemsize	Number of bytes used by a single element in the array
a.nbytes	Number of bytes used by entire array (data only)
a.ndim	Number of dimensions in the array
	SHAPE OPERATIONS
a.flat	An iterator to step through array as if it were 1D
a.flatten()	Returns a 1D copy of a multi-dimensional array
a.ravel()	Same as flatten(), but returns a "view" if possible
a.resize(new_size)	Changes the size/shape of an array in place
a.swapaxes(axis1, axis2)	Swaps the order of two axes in an array
a.transpose(*axes)	Swaps the order of any number of array axes
a.T	Shorthand for a.transpose()
a.squeeze()	Removes any length==1 dimensions from an array

Summary of (most) array attributes/methods (2/4)

ENTHOUGHT

	FILL AND COPY
`a.copy()`	Returns a copy of the array
`a.fill(value)`	Fills an array with a scalar value
	CONVERSION/COERCION
`a.tolist()`	Converts array into nested lists of values
`a.tostring()`	Raw copy of array memory into a Python string
`a.astype(dtype)`	Returns array coerced to the given type
`a.byteswap(False)`	Converts byte order (big <->little endian)
`a.view(type_or_dtype)`	Creates a new ndarray that sees the same memory but interprets it as a new datatype (or subclass of ndarray)
	COMPLEX NUMBERS
`a.real`	Returns the real part of the array
`a.imag`	Returns the imaginary part of the array
`a.conjugate()`	Returns the complex conjugate of the array
`a.conj()`	Returns the complex conjugate of the array (same as conjugate)

Summary of (most) array attributes/methods (3/4)

ENTHOUGHT

	SAVING
`a.dump(file)`	Stores binary array data to *file*
`a.dumps()`	Returns a binary pickle of the data as a string
`a.tofile(fid, sep="", format="%s")`	Formatted ASCII output to a file
	SEARCH/SORT
`a.nonzero()`	Returns indices for all non-zero elements in the array
`a.sort(axis=-1)`	Sort the array elements in place, along *axis*
`a.argsort(axis=-1)`	Finds indices for sorted elements, along *axis*
`a.searchsorted(b)`	Finds indices where elements of *b* would be inserted in *a* to maintain order
	ELEMENT MATH OPERATIONS
`a.clip(low, high)`	Limits values in the array to the specified range
`a.round(decimals=0)`	Rounds to the specified number of digits
`a.cumsum(axis=None)`	Cumulative sum of elements along *axis*
`a.cumprod(axis=None)`	Cumulative product of elements along *axis*

Summary of (most) array attributes/methods (4/4)

ENTHOUGHT

REDUCTION METHODS

All the following methods "reduce" the size of the array by 1 dimension by carrying out an operation along the specified axis. If axis is None, the operation is carried out across the entire array.

`a.sum(axis=None)`	Sums values along axis
`a.prod(axis=None)`	Finds the product of all values along axis
`a.min(axis=None)`	Finds the minimum value along axis
`a.max(axis=None)`	Finds the maximum value along axis
`a.argmin(axis=None)`	Finds the index of the minimum value along axis
`a.argmax(axis=None)`	Finds the index of the maximum value along axis
`a.ptp(axis=None)`	Calculates a.max(axis) – a.min(axis)
`a.mean(axis=None)`	Finds the mean (average) value along axis
`a.std(axis=None)`	Finds the standard deviation along axis
`a.var(axis=None)`	Finds the variance along axis
`a.any(axis=None)`	True if any value along axis is non-zero (logical OR)
`a.all(axis=None)`	True if all values along axis are non-zero (logical AND)

Array Creation Functions

ENTHOUGHT

ARANGE

`arange(start,stop=None,step=1,`
`dtype=None)`

Nearly identical to Python's **`range()`**. Creates an array of values in the range [start,stop) with the specified step value. Allows non-integer values for start, stop, and step. Default **`dtype`** is derived from the start, stop, and step values.

```
>>> arange(4)
array([0, 1, 2, 3])
>>> arange(0, 2*pi, pi/4)
array([ 0.000, 0.785, 1.571,
2.356, 3.142, 3.927, 4.712,
5.497])

# Be careful…
>>> arange(1.5, 2.1, 0.3)
array([ 1.5, 1.8, 2.1])
```

ONES, ZEROS

`ones(shape, dtype=float64)`
`zeros(shape, dtype=float64)`

`shape` is a number or sequence specifying the dimensions of the array. If **`dtype`** is not specified, it defaults to **`float64`**.

```
>>> ones((2,3),dtype=float32)
array([[ 1.,  1.,  1.],
   [ 1.,  1.,  1.]],
      dtype=float32)
>>> zeros(3)
array([ 0., 0., 0.])
```

Array Creation Functions (cont.)

ENTHOUGHT

IDENTITY

```
# Generate an n by n identity
# array. The default dtype is
# float64.
>>> a = identity(4)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> a.dtype
dtype('float64')
>>> identity(4, dtype=int)
array([[ 1,  0,  0,  0],
       [ 0,  1,  0,  0],
       [ 0,  0,  1,  0],
       [ 0,  0,  0,  1]])
```

EMPTY AND FILL

```
# empty(shape, dtype=float64,
#       order='C')
>>> a = empty(2)
>>> a
array([1.78021120e-306,
 6.95357225e-308])

# fill array with 5.0
>>> a.fill(5.0)
array([5., 5.])

# alternative approach
# (slightly slower)
>>> a[:] = 4.0
array([4., 4.])
```


Array Creation Functions (cont.)

ENTHOUGHT

LINSPACE

```
# Generate N evenly spaced
# elements between (and
# including) start and
# stop values.
>>> linspace(0,1,5)
array([0.,0.25.,0.5,0.75, 1.0])
```

LOGSPACE

```
# Generate N evenly spaced
# elements on a log scale
# between base**start and
# base**stop (default base=10).
>>> logspace(0,1,5)
array([ 1., 1.77, 3.16,  5.62,
        10.])
```

Array Creation Functions (cont.)

MGRID

```
# Get equally spaced points
# in N output arrays for an
# N-dimensional (mesh) grid.

>>> x,y = mgrid[0:5,0:5]
>>> x
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])
>>> y
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

OGRID

```
# Construct an "open" grid
# of points (not filled in
# but correctly shaped for
# math operations to be
# broadcast correctly).

>>> x,y = ogrid[0:3,0:3]
>>> x
array([[0],
       [1],
       [2]])
>>> y
array([[0, 1, 2]])

>>> print(x+y)
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```


Trig and Other Functions

TRIGONOMETRIC

`sin(x)`	`sinh(x)`
`cos(x)`	`cosh(x)`
`arccos(x)`	`arccosh(x)`
`arctan(x)`	`arctanh(x)`
`arcsin(x)`	`arcsinh(x)`
`arctan2(x,y)`	

OTHERS

`exp(x)`	`log(x)`
`log10(x)`	`sqrt(x)`
`absolute(x)`	`conjugate(x)`
`negative(x)`	`ceil(x)`
`floor(x)`	`fabs(x)`
`hypot(x,y)`	`fmod(x,y)`
`maximum(x,y)`	`minimum(x,y)`

VECTOR OPERATIONS

`dot(x,y)`	`vdot(x,y)`
`inner(x,y)`	`outer(x,y)`
`cross(x,y)`	`kron(x,y)`
`tensordot(x,y[,axis])`	

`hypot(x,y)`

Element by element distance calculation using $\sqrt{x^2 + y^2}$

More Basic Functions

TYPE HANDLING

iscomplexobj	real_if_close	isnan
iscomplex	isscalar	nan_to_num
isrealobj	isneginf	common_type
isreal	isposinf	typename
imag	isinf	
real	isfinite	

SHAPE MANIPULATION

atleast_1d	hstack	hsplit
atleast_2d	vstack	vsplit
atleast_3d	dstack	dsplit
expand_dims	column_stack	split
apply_over_axes		squeeze
apply_along_axis		

OTHER USEFUL FUNCTIONS

fix	unwrap	roots
mod	sort_complex	poly
amax	trim_zeros	any
amin	fliplr	all
ptp	flipud	disp
sum	rot90	unique
cumsum	eye	diff
prod	diag	angle
cumprod	select	extract
		insert

NAN-RELATED FUNCTIONS

nansum	nanmean	nanstd
nanmax	nanmin	nanvar
nanargmax	nanargmin	

Vectorizing Functions

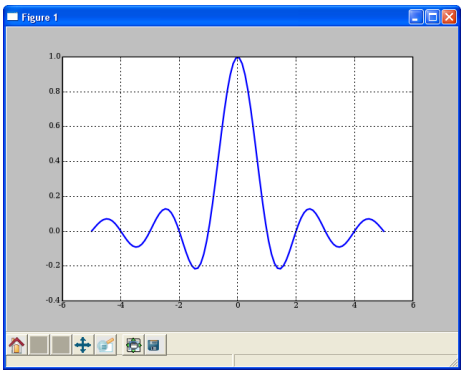
SCALAR SINC FUNCTION

```
# special.sinc already available
# This is just for show.
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        w = pi*x
        return sin(w) / w
```

```
# attempt
>>> x = array((1.3, 1.5))
>>> sinc(x)
ValueError: The truth value of
an array with more than one
element is ambiguous. Use
a.any() or a.all()
```

SOLUTION

```
>>> from numpy import vectorize
>>> vsinc = vectorize(sinc)
>>> vsinc(x)
array([-0.1981, -0.2122])

>>> x2 = linspace(-5, 5, 101)
>>> plot(x2, vsinc(x2))
```



Mathematical Binary Operators

```
a + b → add(a,b)
a - b → subtract(a,b)
a % b → remainder(a,b)
```

```
a * b  → multiply(a,b)
a / b  → divide(a,b)
a ** b → power(a,b)
```

MULTIPLY BY A SCALAR

```
>>> a = array((1,2))
>>> a*3.
array([3., 6.])
```

ELEMENT BY ELEMENT ADDITION

```
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

ADDITION USING AN OPERATOR FUNCTION

```
>>> add(a,b)
array([4, 6])
```

IN-PLACE OPERATION

```
# Overwrite contents of a.
# Saves array creation
# overhead.
>>> add(a,b,a) # a += b
array([4, 6])
>>> a
array([4, 6])
```


Comparison and Logical Operators

```
equal         (==)
greater_equal (>=)
logical_and
logical_not
```

```
not_equal   (!=)
less        (<)
logical_or
```

```
greater     (>)
less_equal  (<=)
logical_xor
```

2-D EXAMPLE

```
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[True, True, False, True],
       [False, True, True, True]])

# functional equivalent
>>> equal(a,b)
array([[True, True, False, True],
       [False, True, True, True]])
```

Be careful with if statements involving numpy arrays. To test for equality of arrays, don't do:

```
if a == b:
```

Rather, do:

```
if all(a==b):
```

For floating point,

```
if allclose(a,b):
```

is even better.

Bitwise Operators

bitwise_and	(&)	invert	(~)	right_shift	(>>)
bitwise_or	(|)	bitwise_xor	(^)	left_shift	(<<)

BITWISE EXAMPLES

```
>>> a = array((1,2,4,8))
>>> b = array((16,32,64,128))
>>> bitwise_or(a,b)
array([ 17,  34,  68, 136])

# bit inversion
>>> a = array((1,2,3,4), uint8)
>>> invert(a)
array([254, 253, 252, 251], dtype=uint8)

# left shift operation
>>> left_shift(a,3)
array([ 8, 16, 24, 32], dtype=uint8)
```

When possible, operation made bitwise are another way to **speed up** computations.

Bitwise and Comparison Together

PRECEDENCE ISSUES

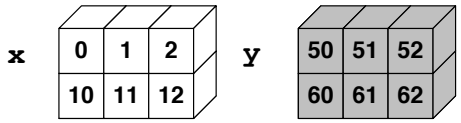
```
# When combining comparisons with bitwise operations,
# precedence requires parentheses around the comparisons.
>>> a = array([1,2,4,8])
>>> b = array([16,32,64,128])
>>> (a > 3) & (b < 100)
array([ False, False, True, False])
```

LOGICAL AND ISSUES

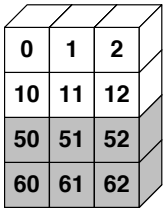
```
# Note that logical AND isn't supported for arrays without
# calling the logical_and function.
>>> a>3 and b<100
Traceback (most recent call last):
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()

# Also, you cannot currently use the "short version" of
# comparison with NumPy arrays.
>>> 2<a<4
Traceback (most recent call last):
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

Array Functions – `concatenate()`

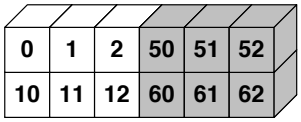
`concatenate((a0,a1,…,aN),axis=0)`
The input arrays `(a0,a1,…,aN)` are concatenated along the given `axis`. They must have the same shape along every axis *except* the one given.



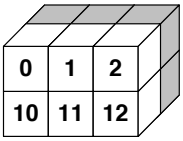

```
>>> concatenate((x,y))
```




```
>>> concatenate((x,y),1)
```



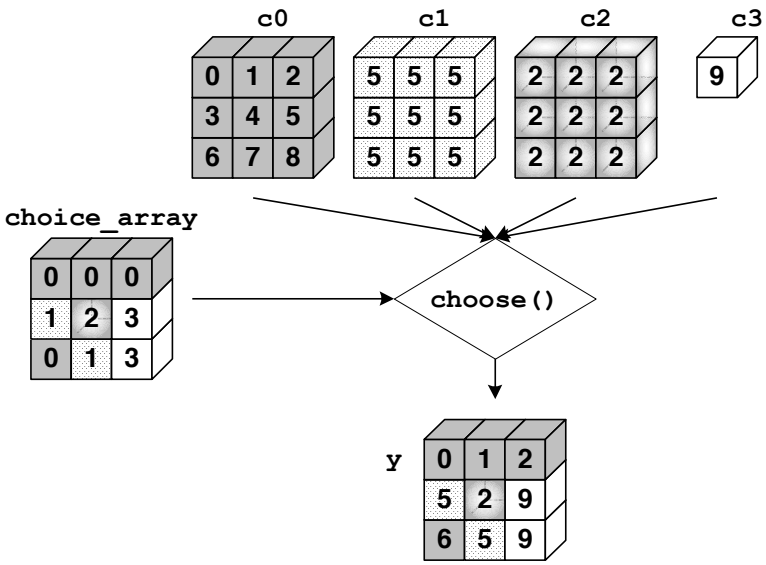

```
>>> array((x,y))
```



See also vstack(), hstack() and dstack() respectively.

Array Functions – `choose()`

```
>>> y = choose(choice_array,(c0,c1,c2,c3))
```



Example - `choose()`

ENTHOUGHT

CLIP LOWER VALUES TO 10

```
>>> a
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22]])

>>> a < 10
array([[True, True, True],
       [False, False, False],
       [False, False, False],
       dtype=bool)

>>> choose(a<10,(a,10))
array([[10, 10, 10],
       [10, 11, 12],
       [20, 21, 22]])
```

CLIP LOWER AND UPPER VALUES

```
>>> lt = a < 10
>>> gt = a > 15
>>> choice = lt + 2 * gt
>>> choice
array([[1, 1, 1],
       [0, 0, 0],
       [2, 2, 2]])
>>> choose(choice,(a,10,15))
array([[10, 10, 10],
       [10, 11, 12],
       [15, 15, 15]])
```


Universal Function Methods

ENTHOUGHT

The mathematical, comparative, logical, and bitwise operators *op* that take two arguments (binary operators) have special methods that operate on arrays:

```
op.reduce(a,axis=0)
op.accumulate(a,axis=0)
op.outer(a,b)
op.reduceat(a,indices)
```

op.reduce()

`op.reduce(a)` applies `op` to all the elements in a 1-D array `a` reducing it to a single value.

For example:

$$y = \texttt{add.reduce(a)}$$
$$= \sum_{n=0}^{N-1} a[n]$$
$$= a[0] + a[1] + \ldots + a[N-1]$$

ADD EXAMPLE

```
>>> a = array([1,2,3,4])
>>> add.reduce(a)
10
```

STRING LIST EXAMPLE

```
>>> a = array(['ab','cd','ef'],
...           dtype=object)
>>> add.reduce(a)
'abcdef'
```

LOGICAL OP EXAMPLES

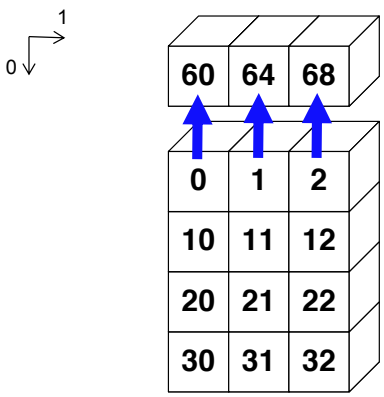
```
>>> a = array([1,1,0,1])
>>> logical_and.reduce(a)
False
>>> logical_or.reduce(a)
True
```


op.reduce()

For multidimensional arrays, `op.reduce(a,axis)` applies `op` to the elements of `a` along the specified `axis`. The resulting array has dimensionality one less than `a`. The default value for `axis` is 0.

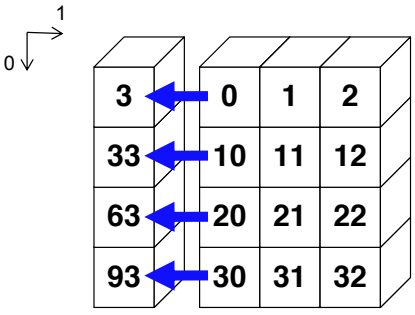
SUM COLUMNS BY DEFAULT

```
>>> add.reduce(a)
array([60, 64, 68])
```



SUMMING UP EACH ROW

```
>>> add.reduce(a,1)
array([ 3, 33, 63, 93])
```



op.accumulate()

op.accumulate(a) creates a new array containing the intermediate results of the **reduce** operation at each element in **a**.

For example:

```
y = add.accumulate(a)
```

$$= \left[\sum_{n=0}^{0} a[n], \sum_{n=0}^{1} a[n], \ldots, \sum_{n=0}^{N-1} a[n] \right]$$

ADD EXAMPLE

```
>>> a = array([1,2,3,4])
>>> add.accumulate(a)
array([ 1,  3,  6, 10])
```

STRING LIST EXAMPLE

```
>>> a = array(['ab','cd','ef'],
...           dtype=object)
>>> add.accumulate(a)
array([ab,abcd,abcdef],
      dtype=object)
```

LOGICAL OP EXAMPLES

```
>>> a = array([1,1,0])
>>> logical_and.accumulate(a)
array([True, True, False])
>>> logical_or.accumulate(a)
array([True, True, True])
```


op.reduceat()

op.reduceat(a,indices) applies `op` to ranges in the 1-D array `a` defined by the values in `indices`. The resulting array has the same length as `indices`.

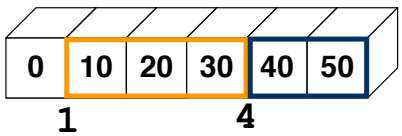
For example:

```
y = add.reduceat(a, indices)
```

$$y[i] = \sum_{n=indices[i]}^{indices[i+1]} a[n]$$

EXAMPLE

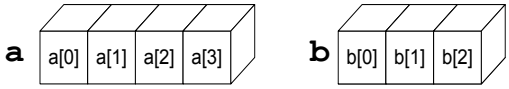
```
>>> a = array([0,10,20,30,
...           40,50])
>>> indices = array([1,4])
>>> add.reduceat(a,indices)
array([60, 90])
```



For multidimensional arrays, `reduceat()` is always applied along the *last* axis (sum of rows for 2-D arrays). This is different from the default for `reduce()` and `accumulate()`.

op.outer()

`op.outer(a,b)` forms all possible combinations of elements between `a` and `b` using `op`. The shape of the resulting array results from concatenating the shapes of `a` and `b`. (Order matters.)




```
>>> add.outer(a,b)
```

a[0]+b[0]	a[0]+b[1]	a[0]+b[2]
a[1]+b[0]	a[1]+b[1]	a[1]+b[2]
a[2]+b[0]	a[2]+b[1]	a[2]+b[2]
a[3]+b[0]	a[3]+b[1]	a[3]+b[2]

```
>>> add.outer(b,a)
```

b[0]+a[0]	b[0]+a[1]	b[0]+a[2]	b[0]+a[3]
b[1]+a[0]	b[1]+a[1]	b[1]+a[2]	b[1]+a[3]
b[2]+a[0]	b[2]+a[1]	b[2]+a[2]	b[2]+a[3]

Array Broadcasting

NumPy arrays of different dimensionality can be combined in the same expression. Arrays with smaller dimension are ***broadcasted*** to match the larger arrays, *without copying data*. Broadcasting has **two rules**.

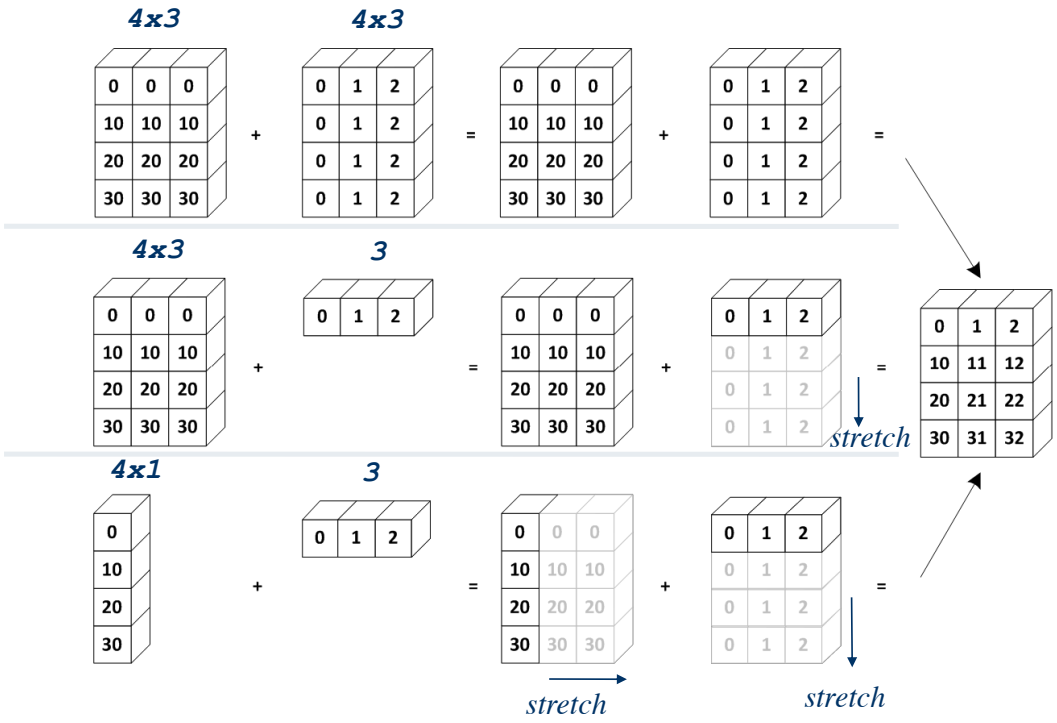
RULE 1: PREPEND ONES TO SMALLER ARRAYS' SHAPE

```
In [3]: a = ones((3, 5)) # a.shape == (3, 5)
In [4]: b = ones((5,)) # b.shape == (5,)
In [5]: b.reshape(1, 5) # result is a (1,5)-shaped array.
In [6]: b[newaxis, :] # equivalent, more concise.
```

RULE 2: DIMENSIONS OF SIZE 1 ARE REPEATED WITHOUT COPYING

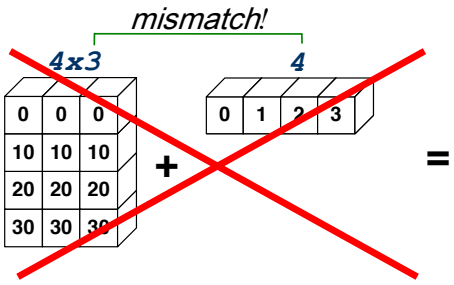
```
In [7]: c = a + b # c.shape == (3, 5)
  is logically equivalent to...
In [8]: tmp_b = b.reshape(1, 5)
In [9]: tmp_b_repeat = tmp_b.repeat(3, axis=0)
In [10]: c = a + tmp_b_repeat
# But broadcasting makes no copies of "b"s data!
```

Array Broadcasting



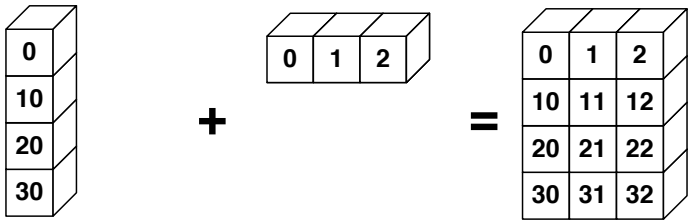
Broadcasting Rules

The *trailing* axes of either arrays must be 1 or both must have the same size for broadcasting to occur. Otherwise, a `"ValueError: shape mismatch: objects cannot be broadcast to a single shape"` exception is thrown.



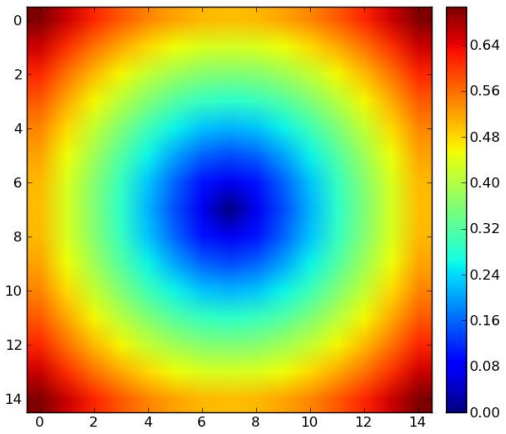
Broadcasting in Action

```
>>> a = array((0,10,20,30))
>>> b = array((0,1,2))
>>> y = a[:, newaxis] + b
```



Application: Distance from Center

```
In [1]: a = linspace(0, 1, 15) - 0.5
In [2]: b = a[:, newaxis] # b.shape == (15, 1)
In [3]: dist2 = a**2 + b**2 # broadcasting sum.
In [4]: dist = sqrt(dist2)
In [5]: imshow(dist); colorbar()
```



Broadcasting's Usefulness

Broadcasting can often be used to replace needless data replication inside a NumPy array expression.

np.meshgrid() – use **newaxis** appropriately in broadcasting expressions.

np.repeat() – broadcasting makes repeating an array along a dimension of size 1 unnecessary.

MESHGRID: COPIES DATA

```
In [3]: x, y = \
   meshgrid([1,2],[3,4,5])
In [4]: z = x + y
```

BROADCASTING: NO COPIES

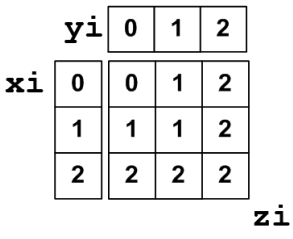
```
In [5]: x = array([1,2])
In [6]: y = array([3,4,5])
In [7]: z = \
   x[newaxis,:] + y[:,newaxis]
```


Broadcasting Indices

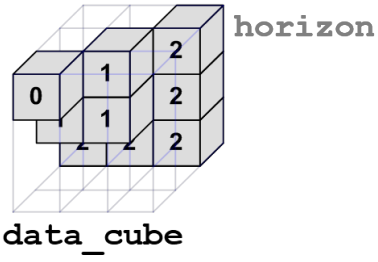
Broadcasting can also be used to slice elements from different "depths" in a 3-D (or any other shape) array. This is a *very* powerful feature of indexing.

```
>>> xi,yi = ogrid[:3,:3]
>>> zi = array([[0, 1, 2],
                [1, 1, 2],
                [2, 2, 2]])
>>> horizon = data_cube[xi,yi,zi]
```

Indices



Selected Data



"Structured" Arrays

```
# "Data structure" (dtype) that describes the fields and
# type of the items in each array element.
>>> particle_dtype = dtype([('mass','float32'), ('velocity', 'float32')])
# This must be a list of tuples.
>>> particles = array([(1,1), (1,2), (2,1), (1,3)],
                      dtype=particle_dtype)
>>> print(particles)
[(1.0, 1.0) (1.0, 2.0) (2.0, 1.0) (1.0, 3.0)]
# Retrieve the mass for all particles through indexing.
>>> print(particles['mass'])
[ 1.  1.  2.  1.]
# Retrieve particle 0 through indexing.
>>> particles[0]
(1.0, 1.0)
# Sort particles in place, with velocity as the primary field and
# mass as the secondary field.
>>> particles.sort(order=('velocity','mass'))
>>> print(particles)
[(1.0, 1.0) (2.0, 1.0) (1.0, 2.0) (1.0, 3.0)]

# See demo/multitype_array/particle.py.
```


"Structured" Arrays

Elements of an array can be any fixed-size data structure!

```
name char[10]
age  int
weight double
```

Brad	Jane	John	Fred
33	25	47	54
135.0	105.0	225.0	140.0
Henry	George	Brian	Amy
29	61	32	27
154.0	202.0	137.0	187.0
Ron	Susan	Jennifer	Jill
19	33	18	54
188.0	135.0	88.0	145.0

EXAMPLE

```
>>> from numpy import dtype, empty
# structured data format
>>> fmt = dtype([('name', 'S10'),
                 ('age', int),
                 ('weight', float)
                ])
>>> a = empty((3,4), dtype=fmt)
>>> a.itemsize
22
>>> a['name'] = [['Brad', … ,'Jill']]
>>> a['age'] = [[33, … , 54]]
>>> a['weight'] = [[135, … , 145]]
>>> print(a)
[[('Brad', 33, 135.0)
  …
  ('Jill', 54, 145.0)]]
```

Nested Datatype

nested.dat

Time	Size	Position				Gain	Samples (2048) ...			
		Az	El	Type	ID					
1172581077060	4108	0.715594	−0.148407	1	4	40	561	1467	997	−30
1172581077091	4108	0.706876	−0.148407	1	4	40	7	591	423	
1172581077123	4108	0.698157	−0.148407	1	4	40	49	−367	−565	−35
1172581077153	4108	0.689423	−0.148407	1	4	40	−55	−953	−1151	−30
1172581077184	4108	0.680683	−0.148407	1	4	40	−719	−1149	−491	38
1172581077215	4108	0.671956	−0.148407	1	4	40	−1503	−683	661	149
1172581077245	4108	0.663232	−0.148407	1	4	40	−2731	−281	2327	291
1172581077276	4108	0.654511	−0.148407	1	4	40	−3493	−159	3277	380
1172581077306	4108	0.645787	−0.148407	1	4	40	−3255	−247	3145	385
1172581077339	4108	0.637058	−0.148407	1	4	40	−2303	−101	2079	247
1172581077370	4108	0.628321	−0.148407	1	4	40	−1495	−553	571	107
1172581077402	4108	0.619599	−0.148407	1	4	40	−955	−1491	−1207	−25
1172581077432	4108	0.61087	−0.148407	1	4	40	−875	−3009	−2987	−93
1172581077463	4108	0.602148	−0.148407	1	4	40	−491	−3681	−4193	−175
1172581077497	4108	0.593438	−0.148407	1	4	40	167	−3501	−4573	−250
1172581077547	4108	0.584696	−0.148407	1	4	40	1007	−2613	−4463	−303
1172581077599	4108	0.575972	−0.148407	1	4	40	1261	−2155	−4299	−339
1172581077650	4108	0.567244	−0.148407	1	4	40	1537	−2633	−4945	−367
1172581077702	4108	0.558511	−0.148407	1	4	40	1105	3701	6139	423

Nested Datatype (cont'd)

The data file can be extracted with the following code:

```
>>> dt = dtype([('time', uint64),
...             ('size', uint32),
...             ('position', [('az', float32),
...                           ('el', float32),
...                           ('region_type', uint8),
...                           ('region_ID', uint16)]),
...             ('gain', uint8),
...             ('samples', int16, 2048)])

>>> data = loadtxt('nested.dat', dtype=dt, skiprows=2)
>>> data['position']['az']
array([ 0.71559399,  0.70687598,  0.69815701,  0.68942302,
        0.68068302, ...], dtype=float32)
```