ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Introduction to Numpy

Enthought, Inc.
www.enthought.com

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

www.enthought.com

Q3-2012

Introduction to Numpy

Enthought, Inc.
www.enthought.com

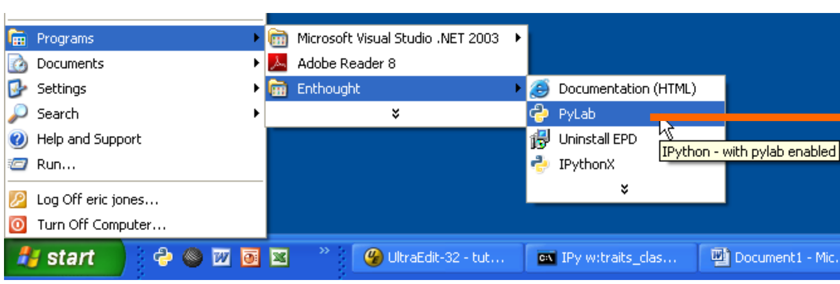
ENTHOUGHT

IPython

An enhanced
interactive Python shell

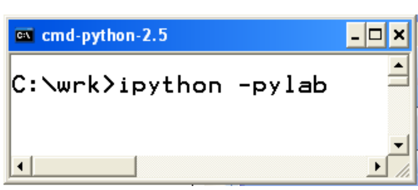
ENTHOUGHT

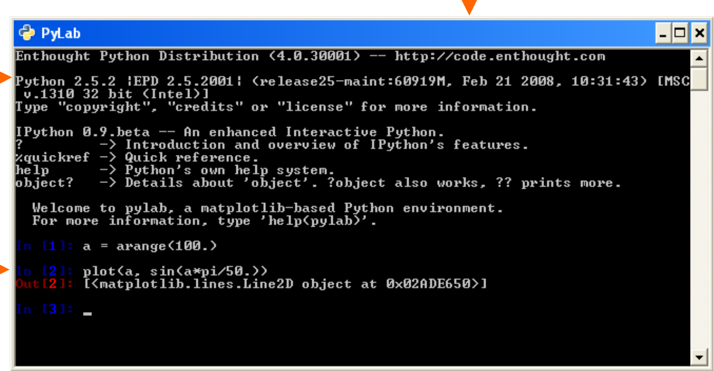
Starting PyLab



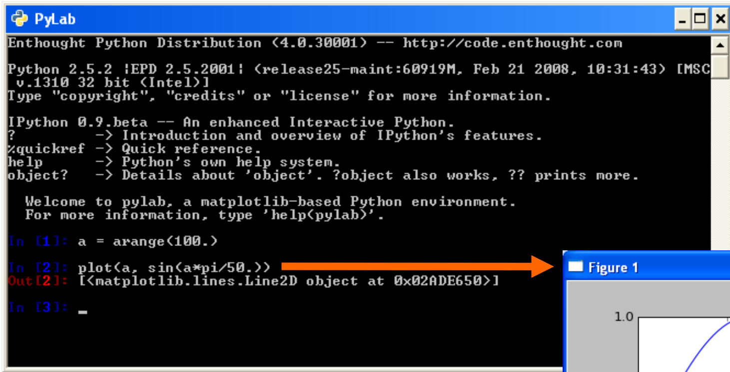
or…

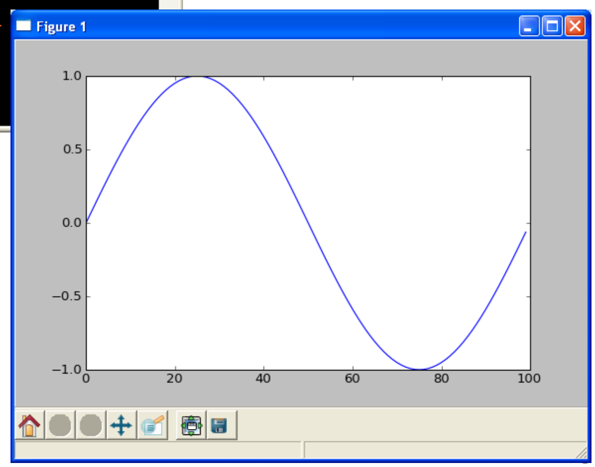
or…





PyLab: Interactive Python Environment





The PyLab mode in IPython handles some gory details behind the scenes. It allows both the Python command interpreter (above) and the GUI plot window (right) to coexist. This involves a bit of multi-threaded magic.

PyLab also imports some handy functions into the command interpreter for user convenience.

IPython

STANDARD PYTHON

```
In [1]: a=1

In [2]: a
Out[2]: 1
```

AVAILABLE VARIABLES

```
In [3]: b = [1,2,3]

# List available variables.
In [4]: %whos
Variable     Type     Data/Length
-----------------------------
a            int      1
b            list     [1, 2, 3]
```

RESET

```
# Remove user defined variables.
In [5]: %reset
In [6]: %whos
Interactive namespace is empty.
```

"%reset" also removes the names imported by PyLab, such as the plot command.

```
In [7]: plot
NameError: name 'plot' is not
defined

# Reload pylab.
In [8]: %pylab
Welcome to pylab,…
```

Directory Navigation in IPython

```
# Change directory (note Unix style forward slashes!)
In [9]: cd c:/python_class/Demos/speed_of_light
c:\python_class\Demos\speed_of_light
```

Tab completion helps you find and type directory and file names.

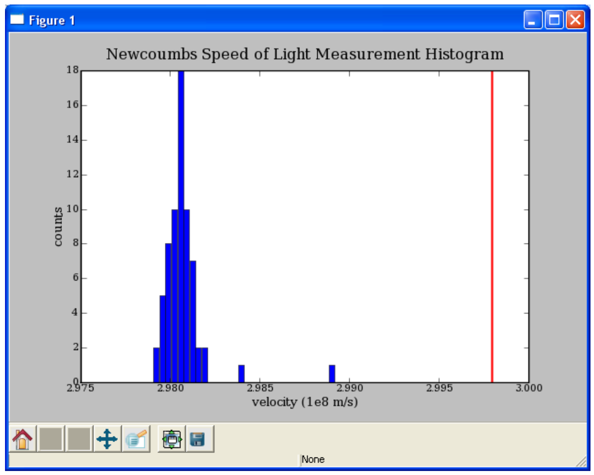
```
# List directory contents (Unix style, not “dir”).
In [10]: ls
 Volume in drive C has no label.
 Volume Serial Number is 5417-593D
 Directory of c:\python_class\Demos\speed_of_light
09/01/2008  02:53 PM  <DIR>          .
09/01/2008  02:53 PM  <DIR>          ..
09/01/2008  02:48 PM         1,188 exercise_speed_of_light.txt
09/01/2008  02:48 PM     2,682,023 measurement_description.pdf
09/01/2008  02:48 PM       187,087 newcomb_experiment.pdf
09/01/2008  02:48 PM         1,312 speed_of_light.dat
09/01/2008  02:48 PM         1,436 speed_of_light.py
09/01/2008  02:48 PM         1,232 speed_of_light2.py
               6 File(s)      2,874,278 bytes
               2 Dir(s)  11,997,437,952 bytes free
```


Directory Bookmarks

```
# Print working directory name (Unix style, not “cd”).
In [11]: pwd
c:\python_class\Demos\speed_of_light

# Bookmark the demo and exercise directories, so we can return
# to them easily.
In [12]: cd ..
c:\python_class\Demos

In [13]: %bookmark demo
In [14]: cd ../Exercises
c:\python_class\Exercises

In [15]: %bookmark exer
In [16]: %bookmark -l
demo -> c:\python_class\Demos
exer -> c:\python_class\Exercises

In [17]: cd demo
(bookmark:demo) -> c:\python_class\Demos
```

Running Scripts in IPython

```
# tab completion
In [11]: %run speed_of_li
speed_of_light.dat  speed_of_light.py

# execute a python file
In [11]: %run speed_of_light.py
```



Function Info

HELP USING ?

```
# Follow a command with '?' to print its documentation.

In [19]: len?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in function len>
Namespace:      Python builtin
Docstring:
    len(object) -> integer

Return the number of items of a sequence or mapping.
```

Function Info

SHOW SOURCE CODE USING ??

```
# Follow a command with '??' to print its source code.
In [43]: squeeze??
def squeeze(a):
    """Remove single-dimensional entries from the shape of a.
    Examples
    --------
    >>> x = array([[[1,1,1],[2,2,2],[3,3,3]]])
    >>> x.shape
    (1, 3, 3)
    >>> squeeze(x).shape
    (3, 3)
    """
    try:
        squeeze = a.squeeze
    except AttributeError:
        return _wrapit(a, 'squeeze')
    return squeeze()
```

`??` can't show the source code for "extension" functions that are implemented in C.

IPython History

HISTORY COMMAND

```
# list previous commands. Use
# 'magic' % because 'hist' is
# histogram function in pylab
In [3]: %hist
1: a=1
2: a
```

INPUT HISTORY

```
# list string from prompt[2]
In [4]: _i2
Out[4]: 'a\n'
```

OUTPUT HISTORY

```
# grab result from prompt[2]
In [5]: _2
Out[5]: 1
```

The up and down arrows scroll through your ipython input history.

Reading Simple Tracebacks

ERROR ADDING AN INTEGER TO A STRING

```
In [9]: 1 + "hello"
---------------------
TypeError        Traceback (most recent call last)
C:\...<ipython-input…> in <module>()
----> 1  1 + "hello"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Location and code where error occurred.

The "type" of error that occurred.

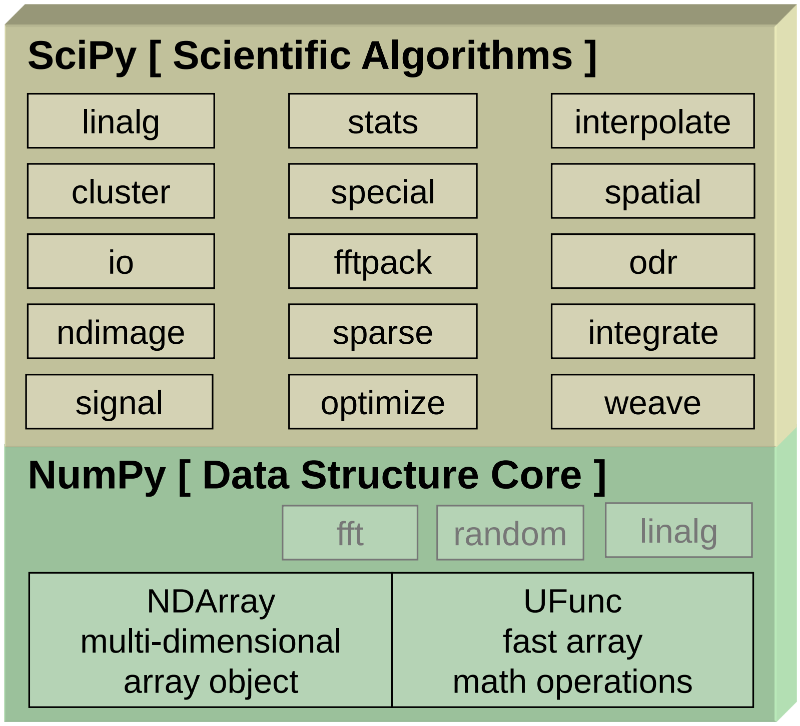
Short message about why it occurred.

ERROR TRYING TO ADD A NON-EXISTENT VARIABLE

```
# Again we fail while adding two variables, but note that the
# traceback tells us that we have a completely different problem.
# In this case, our variable doesn't exists, and thus fails.
In [10]: undefined_var + 1
…
NameError: name 'undefined_var' is not defined
```


NumPy

NumPy and SciPy



NumPy

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

- Offers Matlab-ish capabilities within Python
- NumPy replaces Numeric and Numarray
- Developed by Travis Oliphant
- 46 “committers” to the project (github.com)
- NumPy 1.0 released October, 2006
- NumPy 1.6.1 released July, 2011
- ~25K downloads/month from Sourceforge

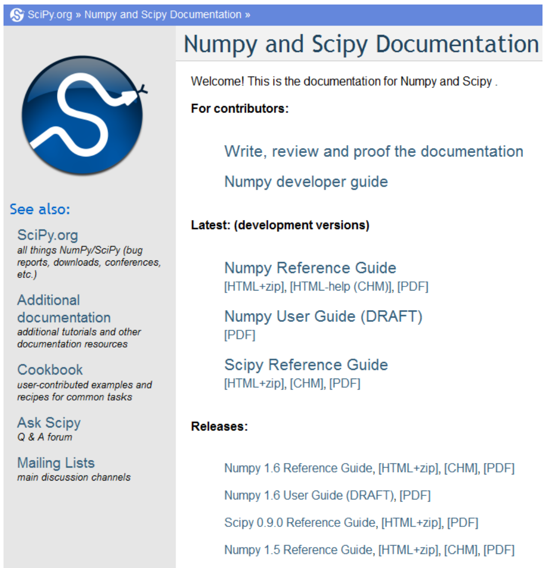
This does not count:
  - Linux distributions that include NumPy
  - Enthought distributions that include NumPy

Helpful Sites

ENTHOUGHT

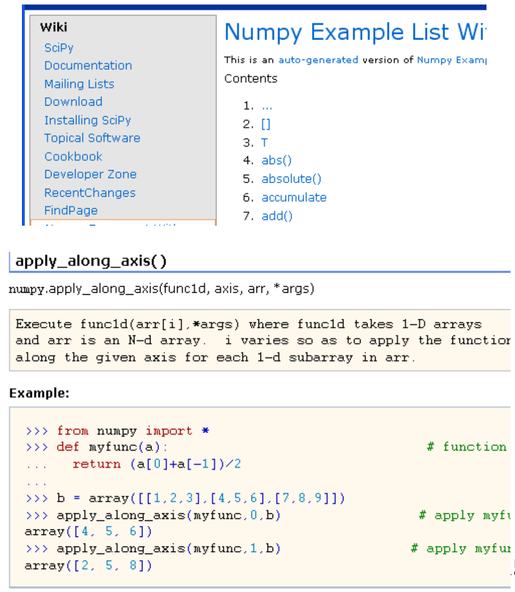
SCIPY DOCUMENTATION PAGE

http://docs.scipy.org/doc



NUMPY EXAMPLES

http://www.scipy.org/Numpy_Example_List_With_Doc



ENTHOUGHT

Getting Started

IMPORT NUMPY

```
In [1]: from numpy import *
In [2]: __version__
Out[2]: 1.6.0
        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.

ENTHOUGHT

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

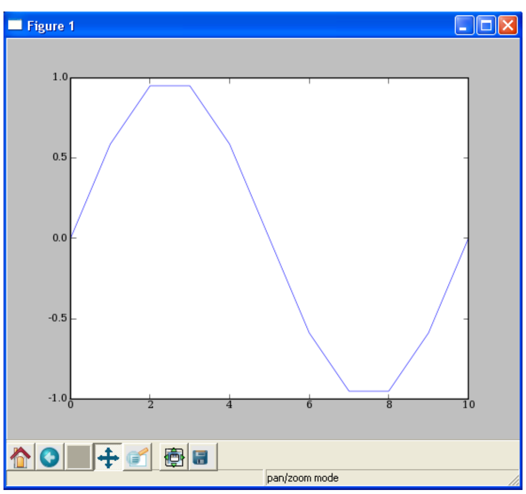
```
# create array from 0 to 10
>>> x = arange(11.)

# 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)
```


ENTHOUGHT

Plotting Arrays

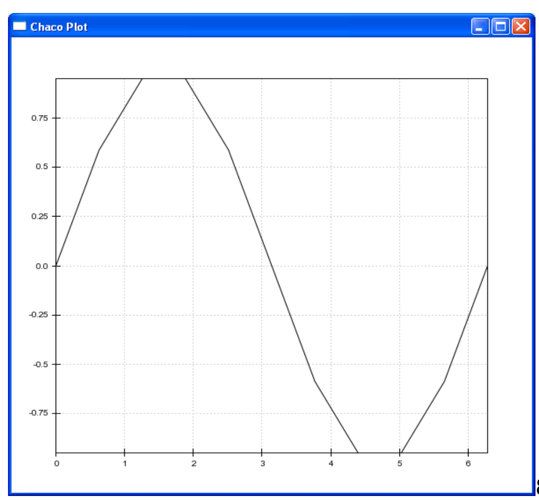
MATPLOTLIB

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



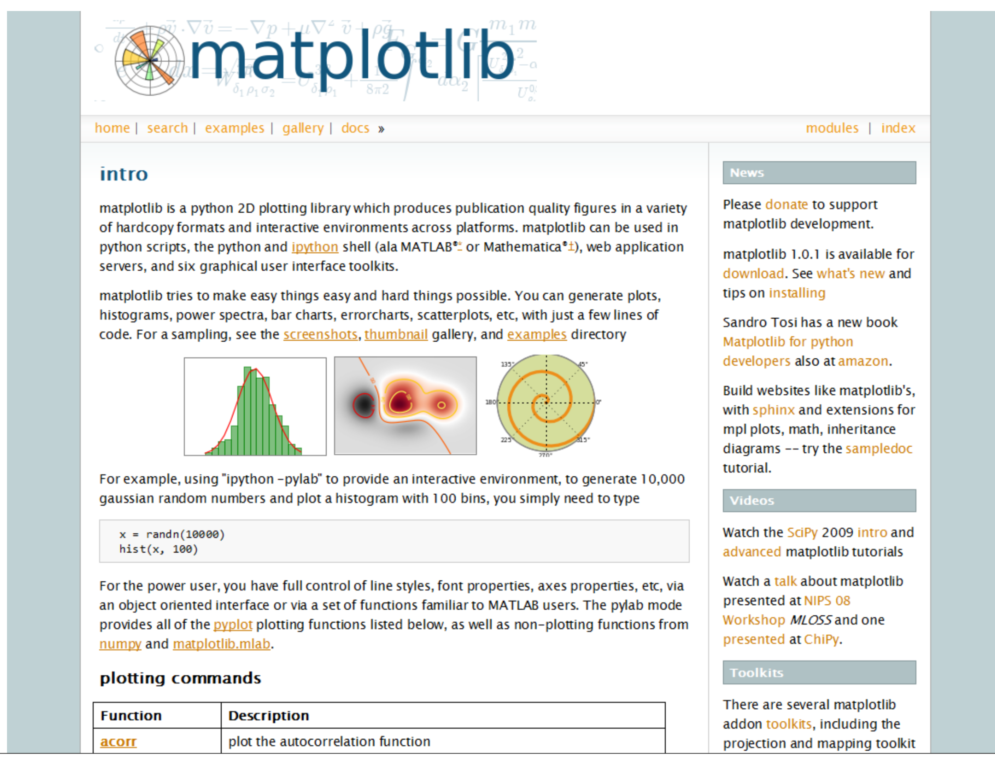
CHACO SHELL

```
>>> from chaco import shell
>>> shell.plot(x,y)
```



Matplotlib Basics
(an interlude)

http://matplotlib.sourceforge.net/

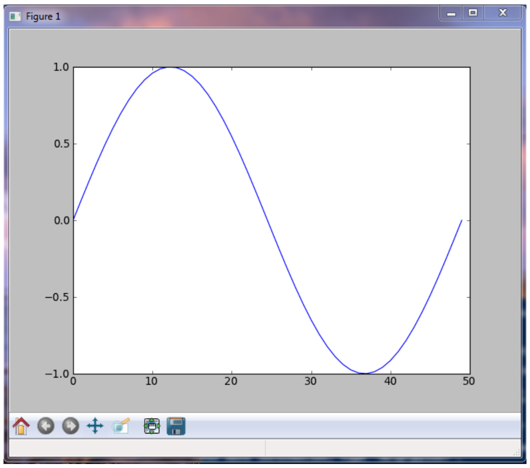


ENTHOUGHT

Line Plots

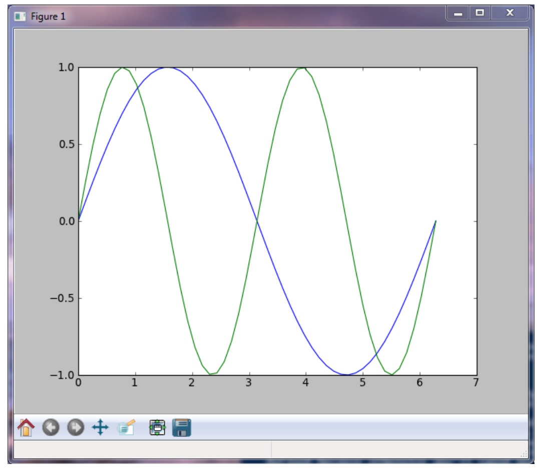
PLOT AGAINST INDICES

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



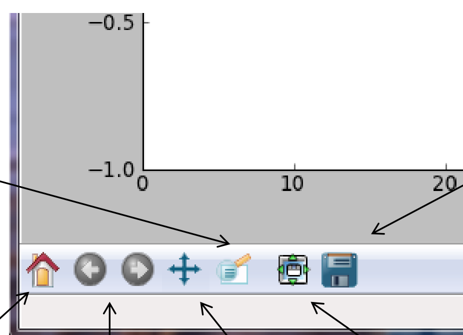
MULTIPLE DATA SETS

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



ENTHOUGHT

Matplotlib Menu Bar



Zoom tool

Save to file (png, svg, eps, pdf, …)

Back to original view

Previous/ next views

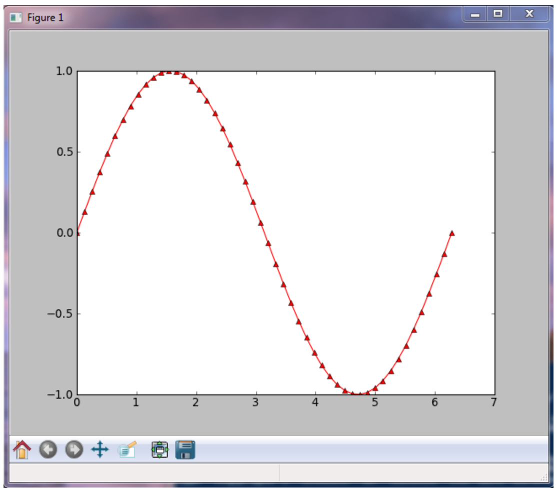
Pan tool

Plot size parameter within window

Line Plots

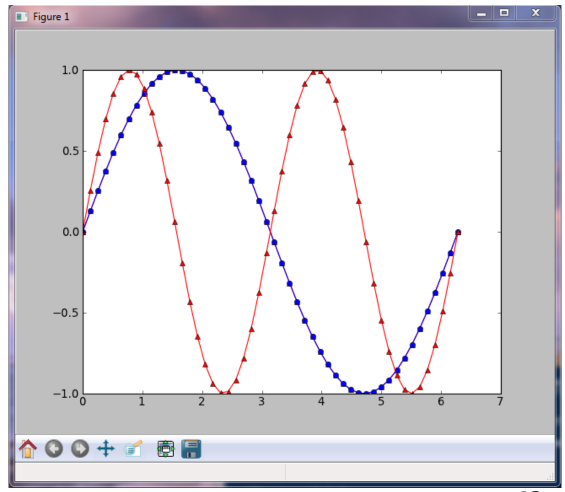
LINE FORMATTING

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



MULTIPLE PLOT GROUPS

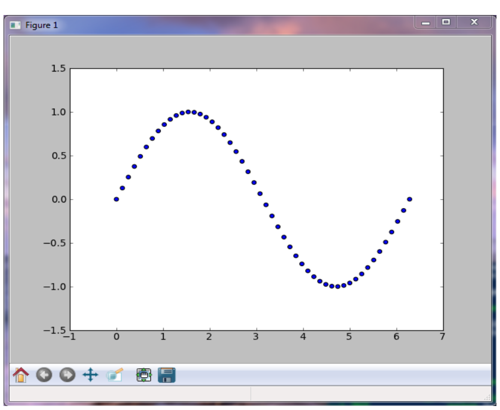
```
>>> plot(x, sin(x), 'b-o',
...      x, sin(2*x), 'r-^')
```



Scatter Plots

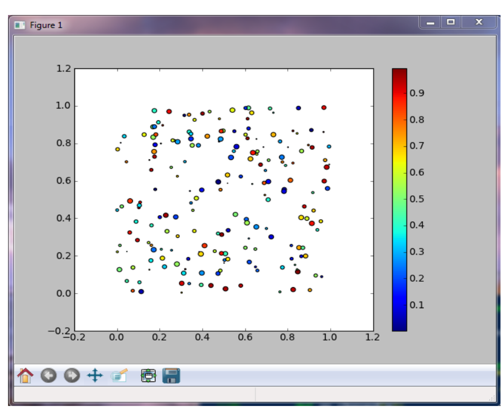
SIMPLE SCATTER PLOT

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



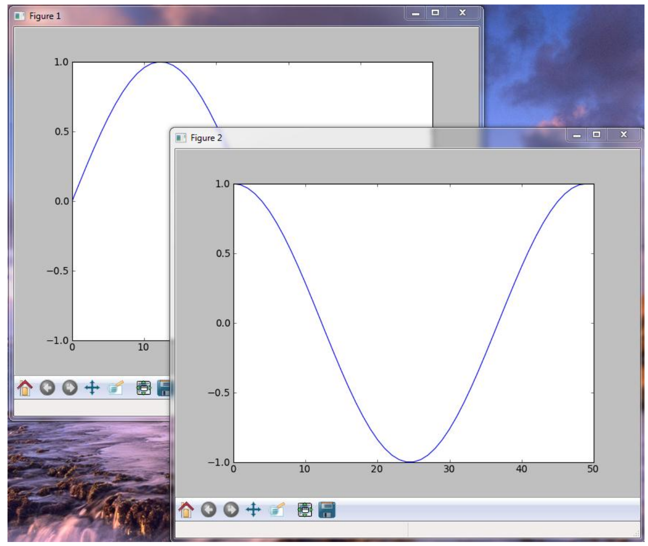
COLORMAPPED SCATTER

```
# 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()
```



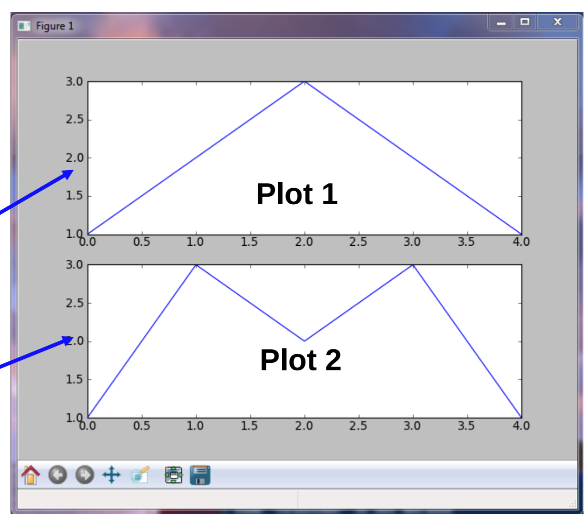
Multiple Figures

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

# Now create a figure
>>> fig1 = figure()
>>> plot(x)

# Now create a new figure.
>>> fig2 = 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)
```



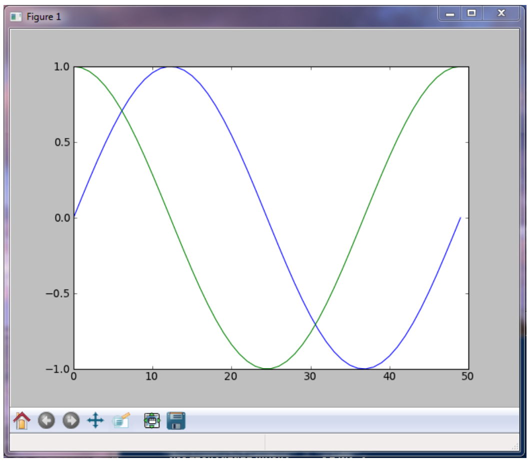
If this is used in a python script, a call to the function show() is required.

ENTHOUGHT

Adding Lines to a Plot

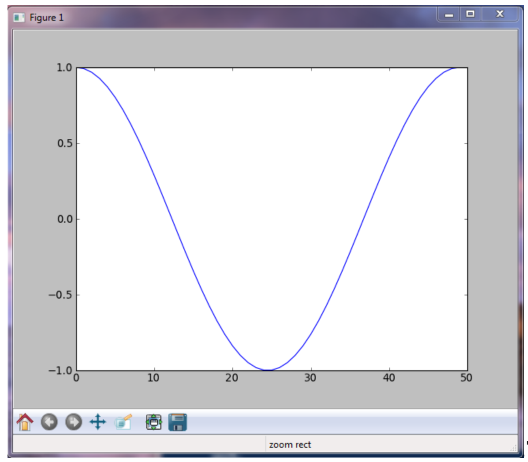
MULTIPLE PLOTS

```
# 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))
```

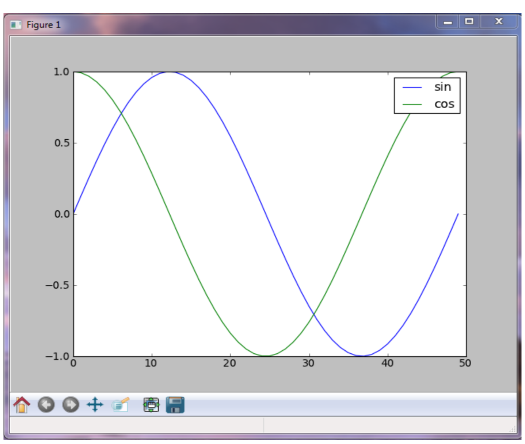


ENTHOUGHT

Legend

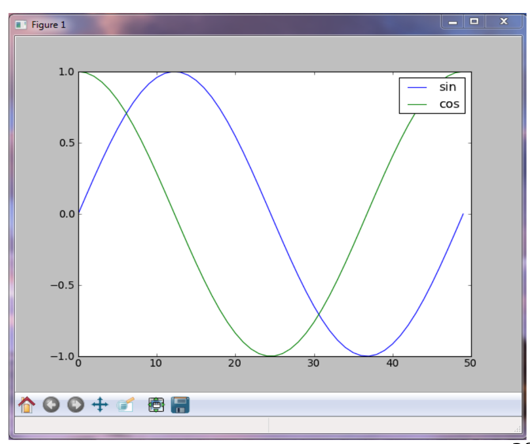
LEGEND LABELS WITH PLOT

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



LABELING WITH LEGEND

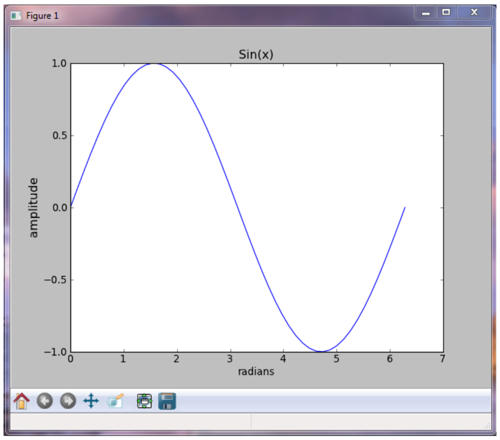
```
# Or as a list in legend().
>>> plot(sin(x))
>>> plot(cos(x))
>>> legend(['sin', 'cos'])
```



Titles and Grid

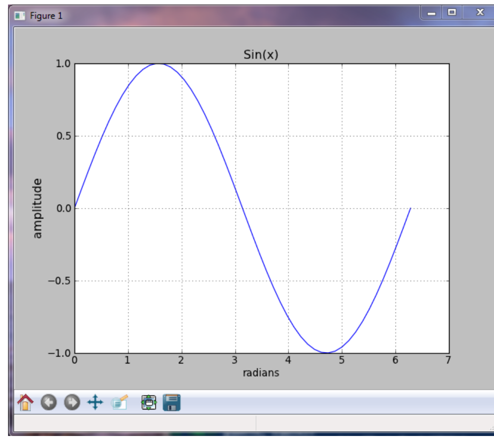
TITLES AND AXIS LABELS

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



PLOT GRID

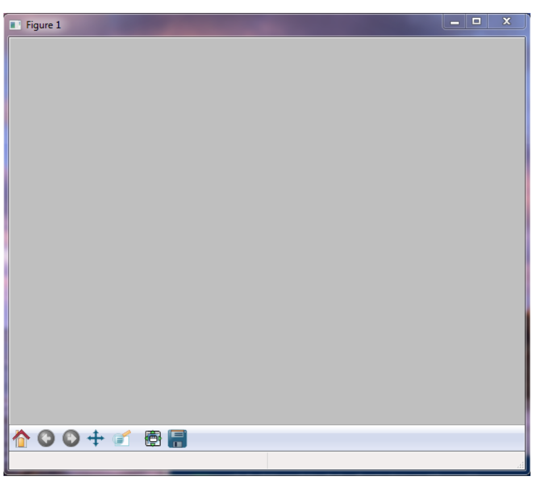
```
# Display gridlines in plot
>>> grid()
```



Clearing and Closing Plots

CLEARING A FIGURE

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

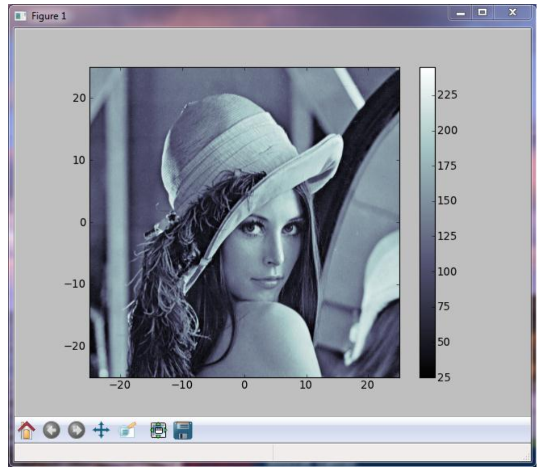


CLOSING PLOT WINDOWS

```
# 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

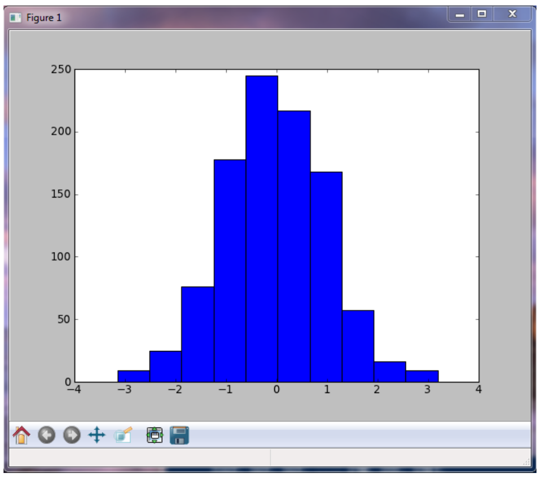
```
# script.py
# In a script, you must call
# the show() command to display
# plots. Call it at the end of
# all your plot commands for
# best performance.
figure()
plot(sin(x))
figure()
plot(cos(x))

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

ENTHOUGHT

Histograms

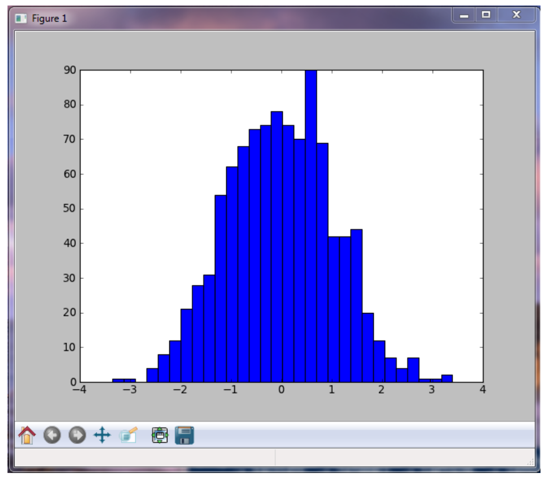
HISTOGRAM

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



HISTOGRAM 2

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

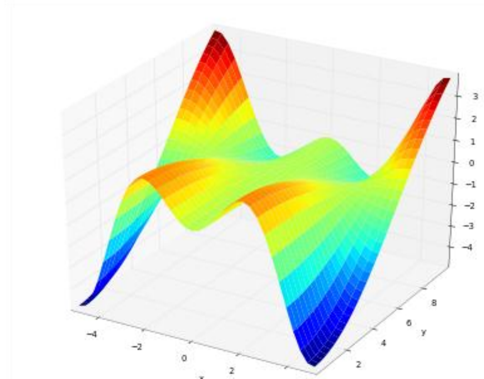


ENTHOUGHT

3D Plots with Matplotlib

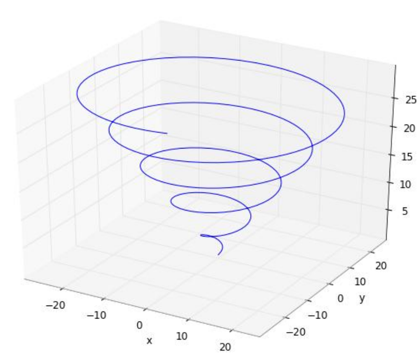
SURFACE PLOT

```
>>> 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')
```



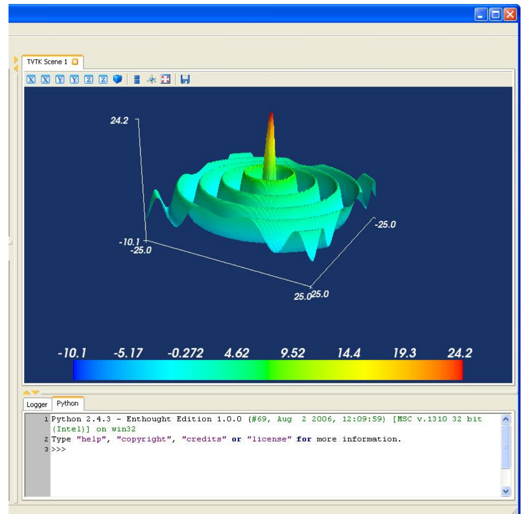
PARAMETRIC CURVE

```
>>> 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’)
```



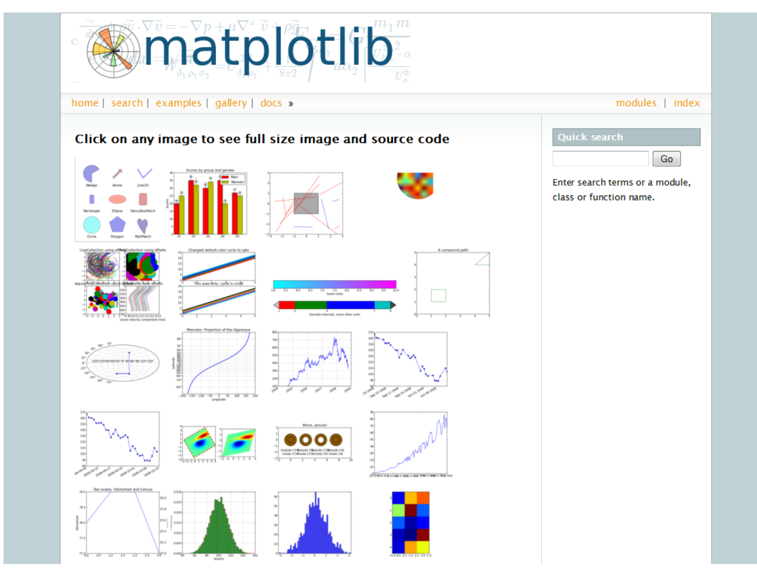
Surface Plots with `mlab`

```
# 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.sourceforge.net/users/screenshots.html
- Gallery (huge): http://matplotlib.sourceforge.net/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
```

ENTHOUGHT

Introducing NumPy Arrays

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
```


ENTHOUGHT

Setting Array Elements

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 LISTS

```
# 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        column
          row
>>> 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])
```

ENTHOUGHT

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
array = loadtxt('Data.txt', skiprows=1,
   dtype=int, delimiter=",",
   usecols = (0,1,2,4), comments = "%")
```

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


ENTHOUGHT

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, EPD	Data files in Geophysics
jpeg, png, …	PIL, scipy.misc.pilutil	Common image formats
fits	pyfits	Image files in Astronomy

ENTHOUGHT

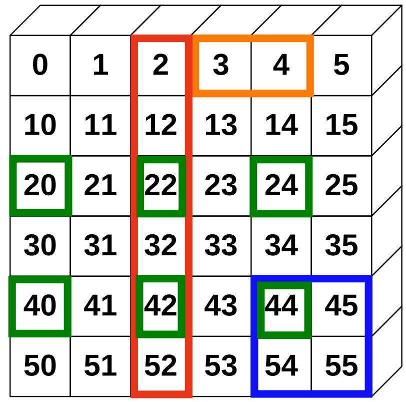
Array Slicing

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]])
```



ENTHOUGHT

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])
```

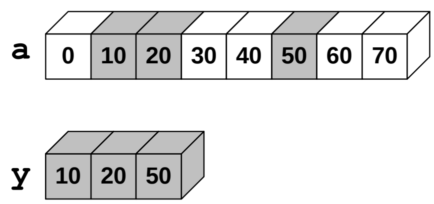
Fancy Indexing

INDEXING BY POSITION

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


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

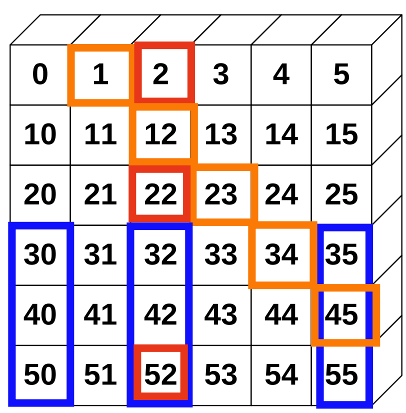
INDEXING WITH BOOLEANS

```
# 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

```
>>> 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])
```

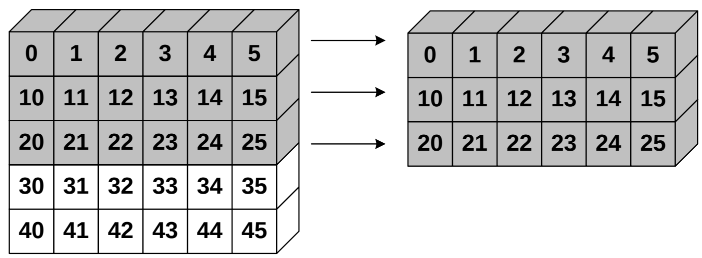


Unlike slicing, fancy indexing creates copies instead of a view into original array.

“Incomplete” Indexing

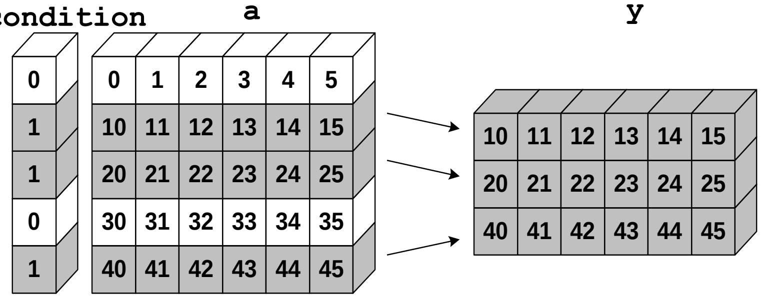
```
>>> y = a[:3]
```

a y




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

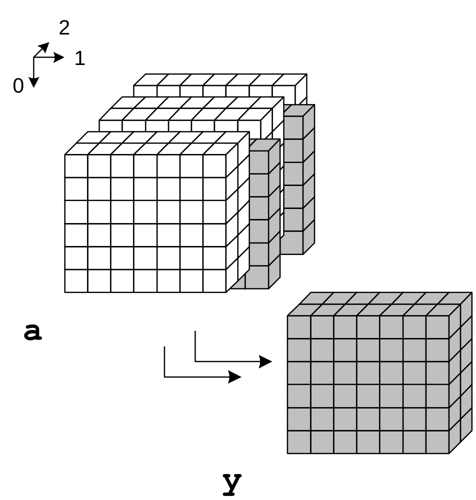
condition a y



3D Example

MULTIDIMENSIONAL

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



Where

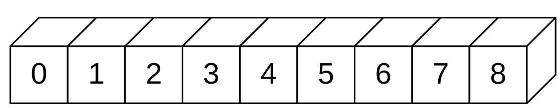
1 DIMENSION

```
# 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

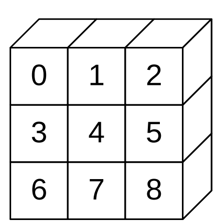
```
# In general, the tuple
# returned is the index of the
# element satisfying the
# condition in each dimension.
>>> a = array([0, 12,  5, 20],
              [1,  2, 11, 15])
>>> loc = where(a>10)
>>> loc
(array([0, 0, 1, 1]),
array([1, 3, 2, 3]))

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


Array Data Structure



Memory block

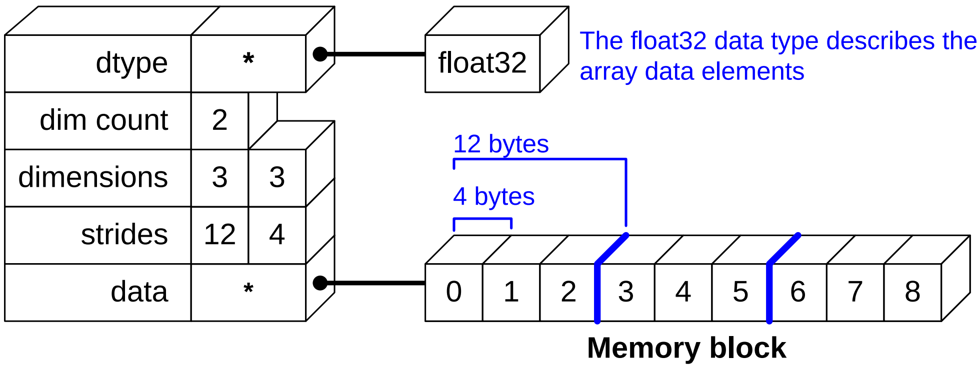
Python View:



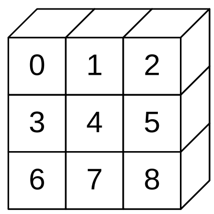
ENTHOUGHT

Array Data Structure

NDArray Data Structure



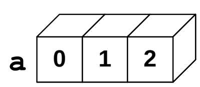
Python View:



ENTHOUGHT

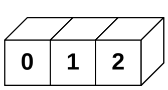
Indexing with newaxis

`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.



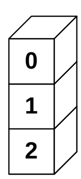
1 X 3

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



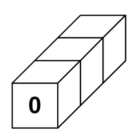
3 X 1

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



3 X 1 X 1

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



ENTHOUGHT

“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                no change
array([[0, 1],
       [2, 3]])
```

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                changed!
array([[10, 1],
       [ 2, 3]])
```


ENTHOUGHT

“(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                changed!
array([[10, 1],
       [ 2, 3]])
```

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)
```

Squeeze

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

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')
```

NumPy dtypes

Basic Type	Available NumPy types	Comments
Boolean	`bool`	Elements are 1 byte in size.
Integer	`int8, int16, int32, int64, int128, int`	int defaults to the size of long in C for the platform.
Unsigned Integer	`uint8, uint16, uint32, uint64, uint128, uint`	uint defaults to the size of unsigned long in C for the platform.
Float	`float16, float32, float64, float,longfloat,`	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`	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`	For example, dtype='S4' would be used for an array of 4-character strings.
Object	`object`	Represent items in array as Python objects.
Records	`void`	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.])
```

ENTHOUGHT

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])
```


ENTHOUGHT

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)

	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)

	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)

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

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.)

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.])
```

ENTHOUGHT

Array Creation Functions (cont.)

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.])
```

ROW SHORTCUT

```
# r_ and c_ are "handy" tools
# (cough hacks…) for creating
# row and column arrays.

# used like arange
# -- real stride value
>>> r_[0:1:.25]
array([ 0., 0.25., 0.5, 0.75])

# used like linspace
# -- complex stride value
>>> r_[0:1:5j]
array([0.,0.25.,0.5,0.75,1.0])

# concatenate elements
>>> r_[(1,2,3),0,0,(4,5)]
array([1, 2, 3, 0, 0, 4, 5])
```


ENTHOUGHT

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]]
```

Matrix Objects

MATRIX CREATION

```
# Matlab-like creation from string
>>> A = mat('1,2,4;2,5,3;7,8,9')
>>> print A
Matrix([[1, 2, 4],
        [2, 5, 3],
        [7, 8, 9]])

# matrix exponents
>>> print A**4
Matrix([[ 6497,  9580,  9836],
        [ 7138, 10561, 10818],
        [18434, 27220, 27945]])

# matrix multiplication
>>> print A*A.I
Matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

BLOCK MATRICES

```
# create a matrix from
# sub-matrices
>>> a = array([[1,2],
               [3,4]])
>>> b = array([[10,20],
               [30,40]])

>>> bmat('a,b;b,a')
matrix([[ 1,  2, 10, 20],
        [ 3,  4, 30, 40],
        [10, 20,  1,  2],
        [30, 40,  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	nansum
prod	diag	nanmax
cumprod	select	nanargmax
diff	extract	nanargmin
angle	insert	nanmin

Vectorizing Functions

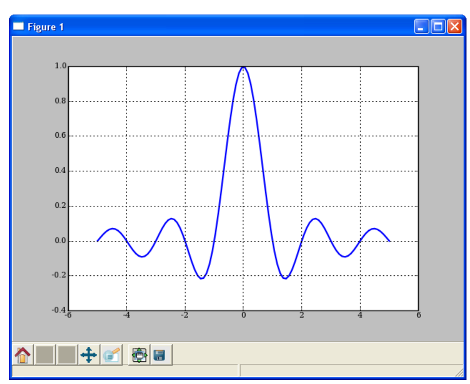
SCALAR SINC FUNCTION

```
# 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  →  multiply(a,b)
a - b →  subtract(a,b)        a / b  →  divide(a,b)
a % b →  remainder(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          (==)    not_equal   (!=)    greater      (>)
greater_equal  (>=)    less        (<)     less_equal   (<=)
logical_and            logical_or          logical_xor
logical_not
```

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(a,shifts)
bitwise_or   (|)   bitwise_xor  (^)   left_shift (a,shifts)
```

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()
```

ENTHOUGHT

Universal Function Methods

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)
```


ENTHOUGHT

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:

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

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
```

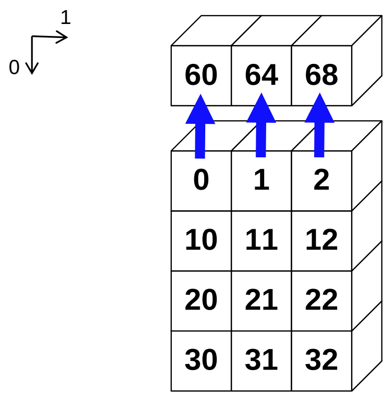
op.reduce()

ENTHOUGHT

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.

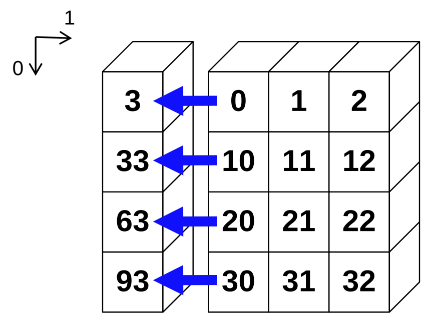
SUM COLUMNS BY DEFAULT

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



SUMMING UP EACH ROW

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



op.accumulate()

ENTHOUGHT

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

For example:

$$\mathtt{y = add.accumulate(a)} = \left[\sum_{n=0}^{0} a[n], \sum_{n=0}^{1} a[n], \ldots, \sum_{n=0}^{N-1} a[n]\right]$$

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()

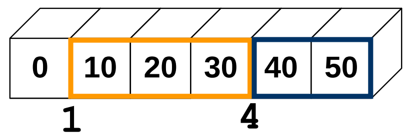
op.reduceat(a,indices)
applies `op` to ranges in the 1-D array `a` defined by the values in `indices`. The resulting array has the same length as `indices`.

For example:
`y = add.reduceat(a, indices)`

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

EXAMPLE

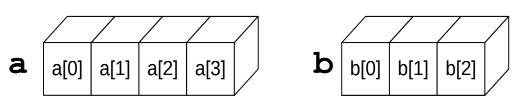
```
>>> a = array([0,10,20,30,
...            40,50])
>>> indices = array([1,4])
>>> add.reduceat(a,indices)
array([60, 90])
```



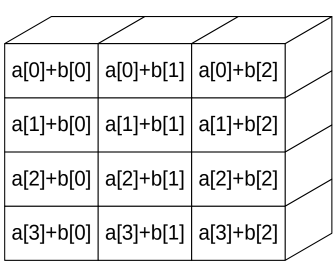
For multidimensional arrays, `reduceat()` is always applied along the *last* axis (sum of rows for 2-D arrays). This is different from the default for `reduce()` and `accumulate()`.

op.outer()

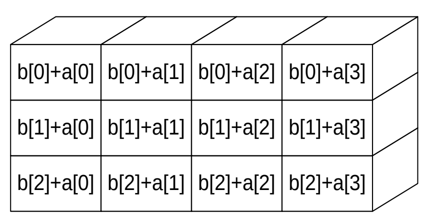
op.outer(a,b) forms all possible combinations of elements between `a` and `b` using `op`. The shape of the resulting array results from concatenating the shapes of `a` and `b`. (Order matters.)




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



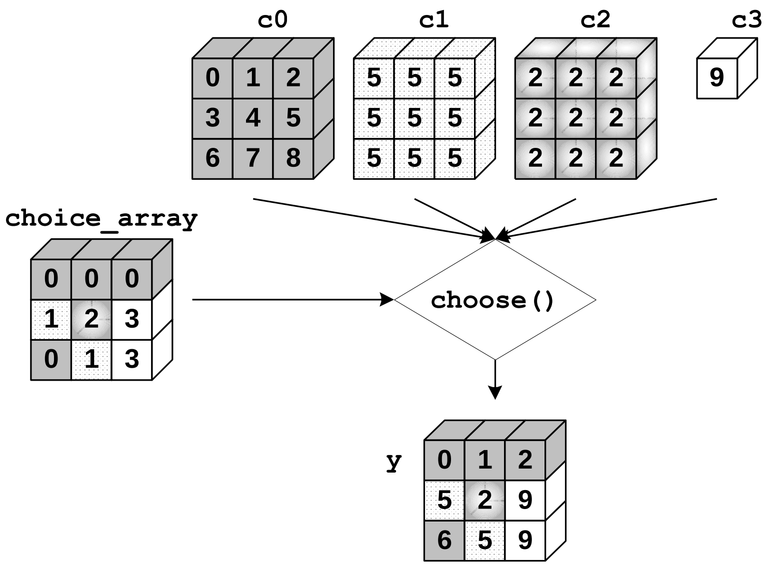

```
>>> add.outer(b,a)
```



Array Functions – choose()

ENTHOUGHT

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



Example - choose()

ENTHOUGHT

CLIP LOWER VALUES TO 10

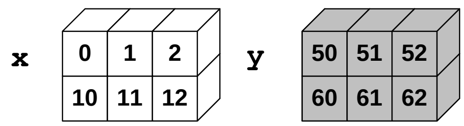
```
>>> a
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22]])

>>> 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

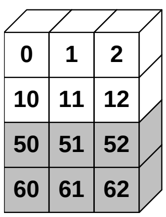
```
>>> lt = a < 10
>>> gt = a > 15
>>> choice = lt + 2 * gt
>>> choice
array([[1, 1, 1],
       [0, 0, 0],
       [2, 2, 2]])
>>> choose(choice,(a,10,15))
array([[10, 10, 10],
       [10, 11, 12],
       [15, 15, 15]])
```

Array Functions – `concatenate()`

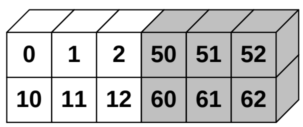
`concatenate((a0,a1,…,aN),axis=0)`
The input arrays **`(a0,a1,…,aN)`** are concatenated along the given **`axis`**. They must have the same shape along every axis *except* the one given.



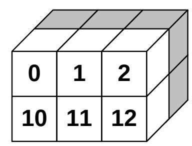
`>>> concatenate((x,y))`



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

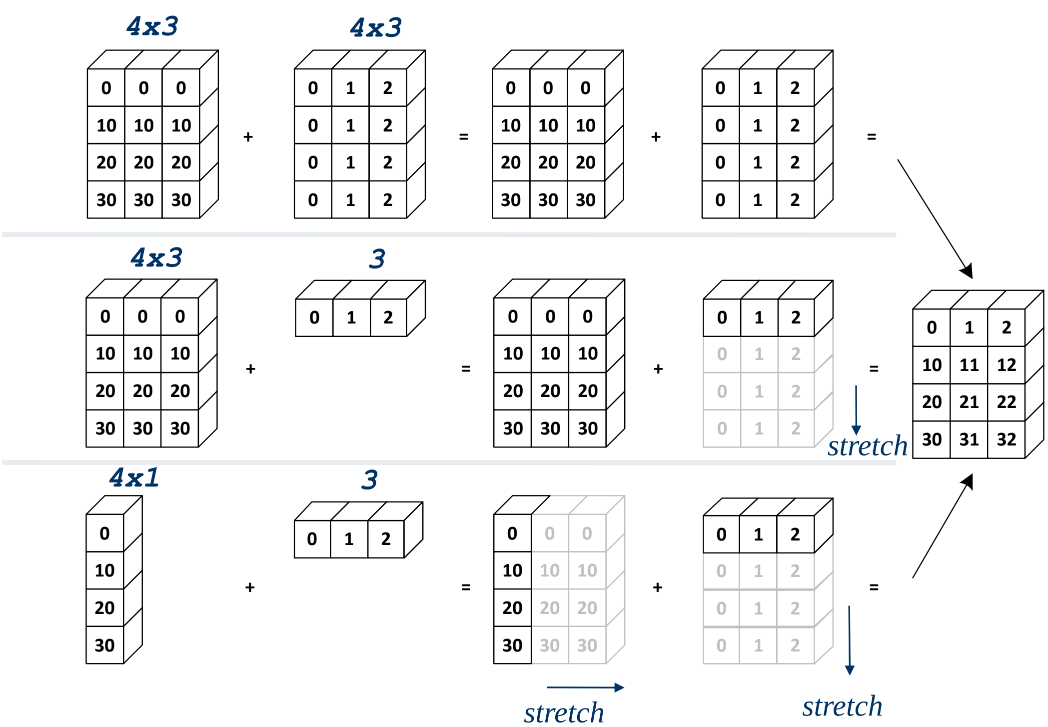


`>>> array((x,y))`



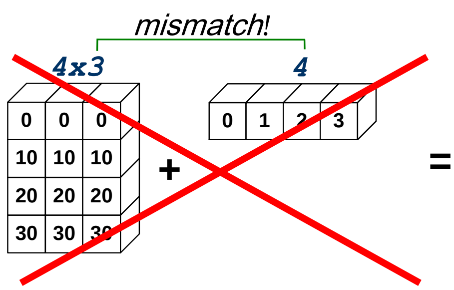
See also vstack(), hstack() and dstack() respectively.

Array Broadcasting



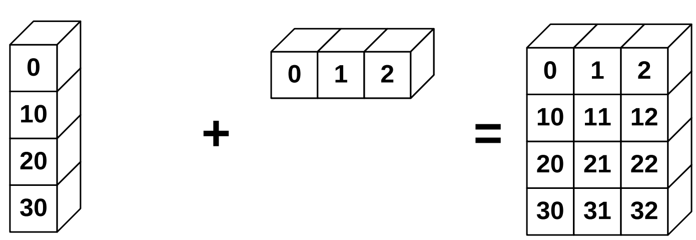
Broadcasting Rules

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.

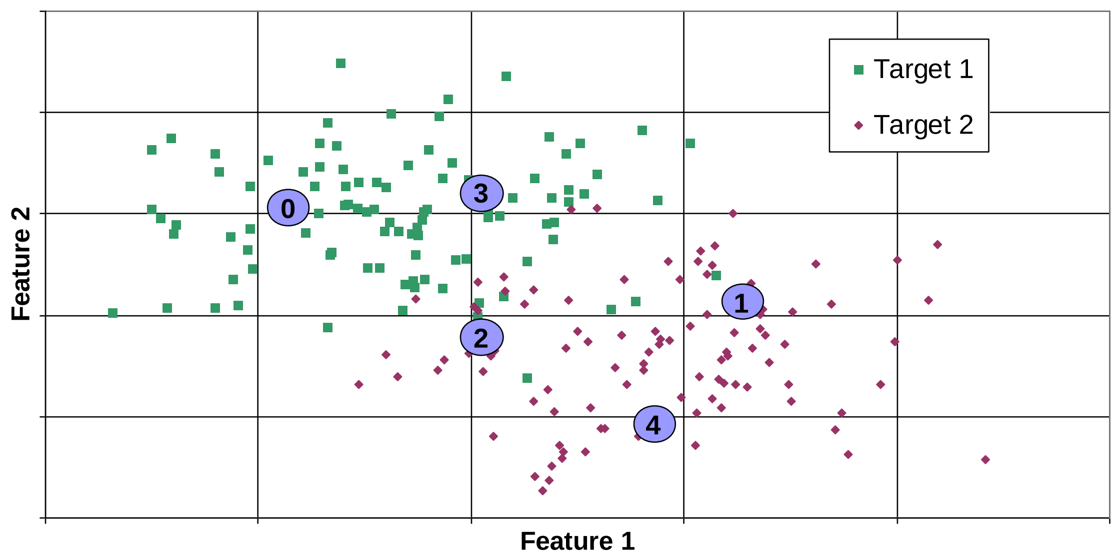


Broadcasting in Action

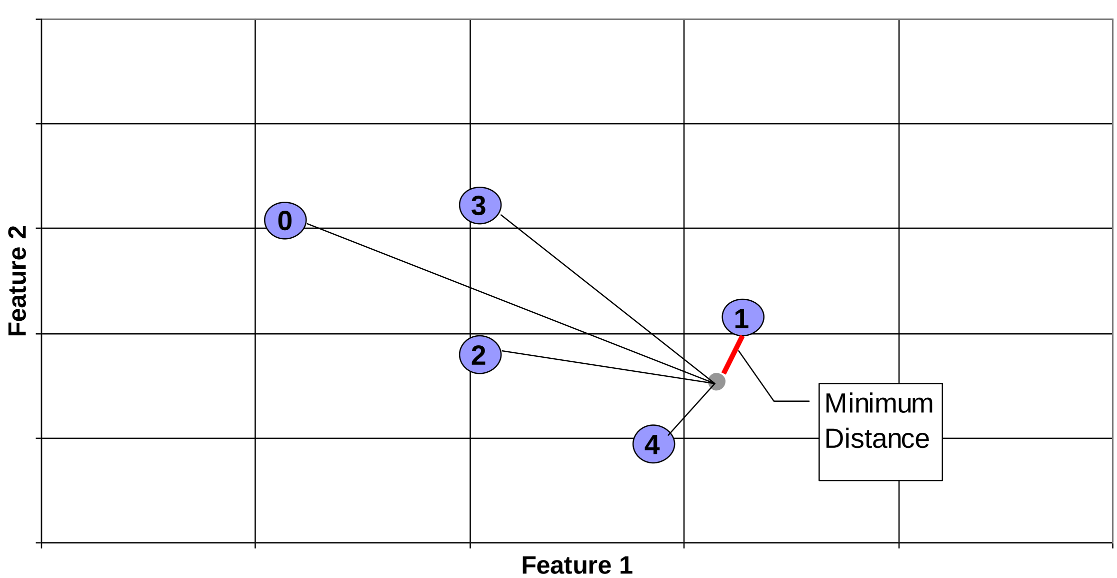
```
>>> a = array((0,10,20,30))
>>> b = array((0,1,2))
>>> y = a[:, newaxis] + b
```



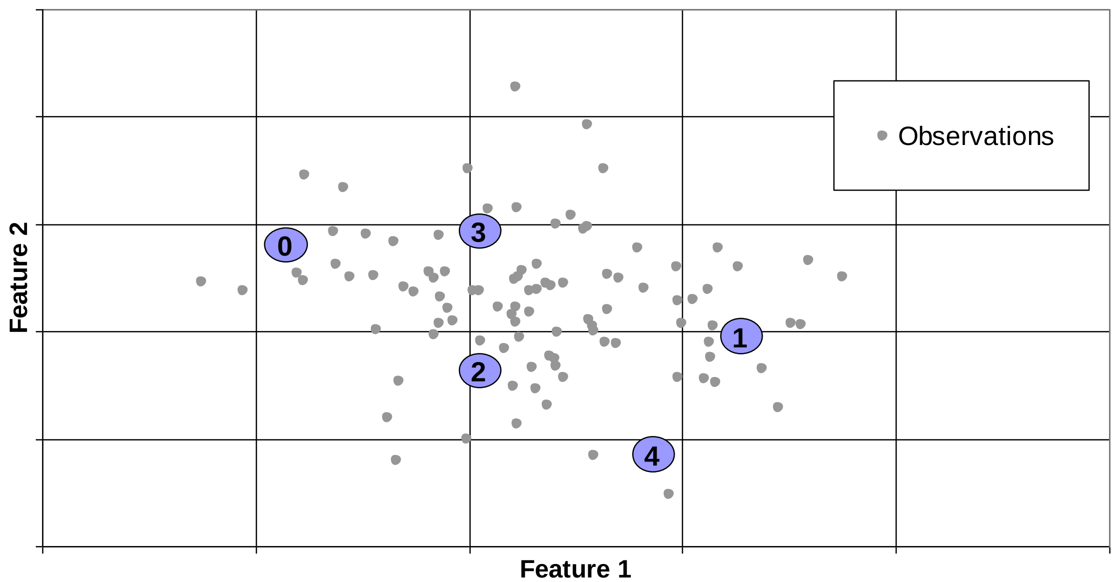
Vector Quantization Example



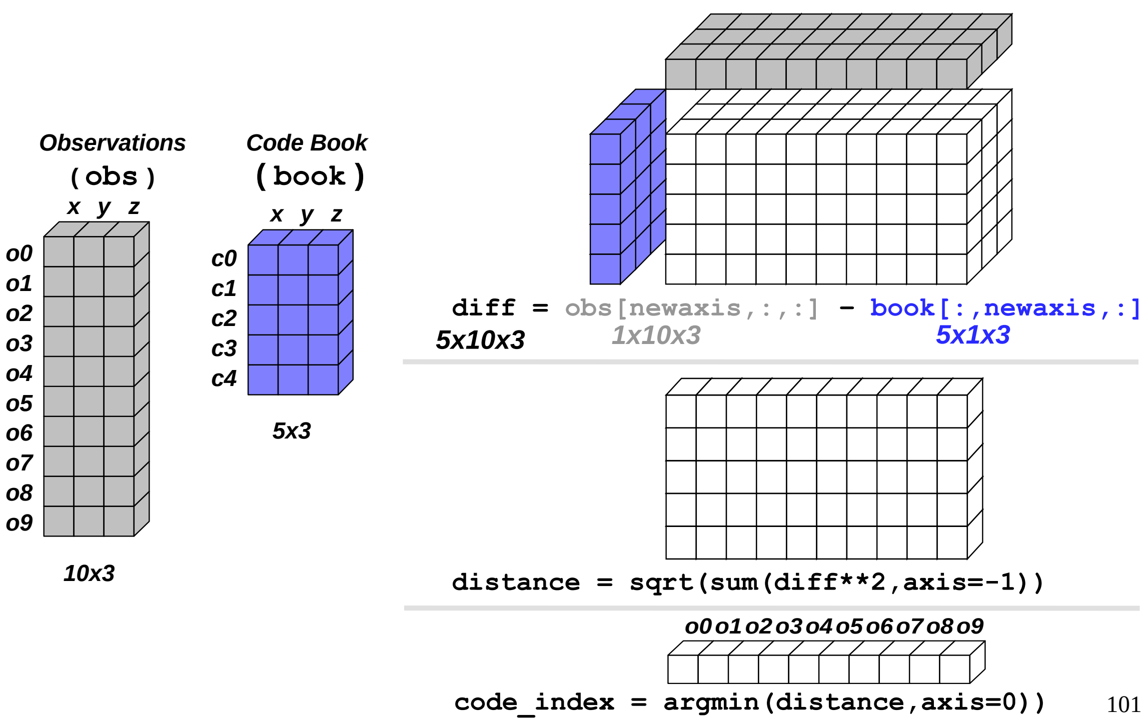
Vector Quantization Example



Vector Quantization Example



Vector Quantization Example



ENTHOUGHT

VQ Speed Comparisons

Method	Run Time (sec)	Speed Up
Matlab 5.3	1.611	-
Python VQ1, double	2.245	0.71
Python VQ1, float	1.138	1.42
Python VQ2, double	1.637	0.98
Python VQ2, float	0.954	1.69
C, double	0.066	24.40
C, float	0.064	24.40

- 4000 observations with 16 features categorized into 40 codes on Pentium III 500 MHz.
- VQ1 uses the technique described on the previous slide verbatim.
- VQ2 applies broadcasting on an observation by observation basis. This turned out to be much more efficient because it is less memory intensive.

ENTHOUGHT

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.

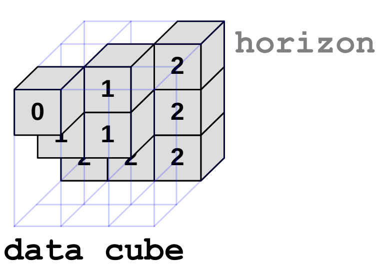
```
>>> xi,yi = ogrid[:3,:3]
>>> zi = array([[0, 1, 2],
                [1, 1, 2],
                [2, 2, 2]])
>>> horizon = data_cube[xi,yi,zi]
```

Indices

xi \ yi	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

zi

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)]]
```

CENTHOUGHT

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	6129	42

CENTHOUGHT

Nested Datatype (cont'd)

The data file can be extracted with the following code:

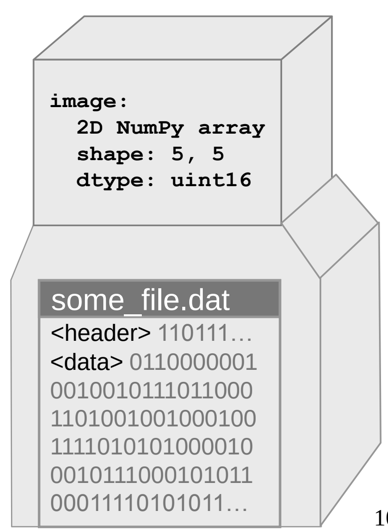
```
>>> dt = dtype([('time', uint64),
...             ('size', uint32),
...             ('position', [('az', float32),
...                           ('el', float32),
...                           ('region_type', uint8),
...                           ('region_ID', uint16)]),
...             ('gain', uint8),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

Memory Mapped Arrays

- Methods for Creating:
  - **memmap**: subclass of ndarray that manages the memory mapping details.
  - **frombuffer**: Create an array from a memory mapped buffer object.
  - **ndarray constructor**: Use the `buffer` keyword to pass in a memory mapped buffer.
- Limitations:
  - Files must be < 2GB on Python 2.4 and before.
  - Files must be < 2GB on 32-bit machines.
  - Python 2.5 and higher on 64 bit machines is theoretically "limited" to 17.2 *billion* GB (17 Exabytes).

Memory Mapped Example

```
# Create a "memory mapped" array where
# the array data is stored in a file on
# disk instead of in main memory.
>>> from numpy import memmap
>>> image = memmap('some_file.dat',
                   dtype=uint16,
                   mode='r+',
                   shape=(5,5),
                   offset=header_size)

# Standard array methods work.
>>> mean_value = image.mean()

# Standard math operations work.
# The resulting scaled_image *is*
# stored in main memory.  It is a
# standard numpy array.
>>> scaled_image = image * .5
```



memmap

The memmap subclass of array handles opening and closing files as well as synchronizing memory with the underlying file system.

```
memmap(filename, dtype=uint8, mode='r+',
       offset=0, shape=None, order=0)
```

filename Name of the underlying file. For all modes, except for 'w+', the file must already exist and contain at least the number of bytes used by the array.

dtype The numpy data type used for the array. This can be a "structured" dtype as well as the standard simple data types.

offset Byte offset within the file to the memory used as data within the array.

mode <see next slide>

shape Tuple specifying the dimensions and size of each dimension in the array. shape=(5,10) would create a 2D array with 5 rows and 10 columns.

order 'C' for row major memory ordering (standard in the C programming language) and 'F' for column major memory ordering (standard in Fortran).

memmap -- mode

The mode setting for memmap arrays is used to set the `access` flag when opening the specified file using the standard `mmap` module.

```
memmap(filename, dtype=uint8, mode='r+',
       offset=0, shape=None, order=0)
```

mode A string indicating how the underlying file should be opened.

'**r**' or '**readonly**': Open an existing file as an array for reading.

'**c**' or '**copyonwrite**': "Copy on write" arrays are "writable" as Python arrays, but they *never* modify the underlying file.

'**r+**' or '**readwrite**': Create a read/write array from an existing file. The file will have "write through" behavior where changes to the array are written to the underlying file. Use the **`flush()`** method to ensure the array is synchronized with the file.

'**w+**' or '**write**': Create the file or overwrite if it exists. The array is filled with zeros and has "write through" behavior similar to 'r+'.

memmap -- write through behavior

```
# Create a memory mapped "write through" file, overwriting it if it exists.
In [66]: q=memmap('new_file.dat',mode='w+',shape=(2,5))
In [67]: q
memmap([[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]], dtype=uint8)
# Print out the contents of the underlying file.  Note: It
# doesn't print because 0 isn't a printable ascii character.
In [68]: !cat new_file.dat

# Now write the ascii value for 'A' (65) into our array.
In [69]: q[:] = ord('A')
In [70]: q
memmap([[65, 65, 65, 65, 65],
        [65, 65, 65, 65, 65]], dtype=uint8)
# Ensure the OS has written the data to the file, and examine
# the underlying file. It is full of 'A's as we hope.
In [71]: q.flush()
In [72]: !cat new_file.dat
AAAAAAAAAA
```

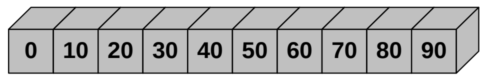

memmap -- copy on write behavior

```
# Create a copy-on-write memory map where the underlying file is never
# modified.  The file must already exist.
# This is a memory efficient way of working with data on disk as arrays but
# ensuring you never modify it.
In [73]: q=memmap('new_file.dat',mode='c',shape=(2,5))
In [74]: q
memmap([[65, 65, 65, 65, 65],
        [65, 65, 65, 65, 65]], dtype=uint8)

# Set values in array to something new.
In [75]: q[1] = ord('B')
In [76]: q
memmap([[65, 65, 65, 65, 65],
        [66, 66, 66, 66, 66]], dtype=uint8)

# Even after calling flush(), the underlying file is not updated.
In [77]: q.flush()
In [78]: !cat new_file.dat
AAAAAAAAAA
```

Using Offsets

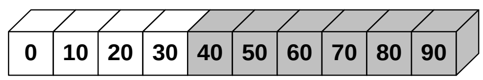
```
# Create a memory mapped array with 10 elements.
In [1]: q=memmap('new_file.dat',mode='w+', dtype=uint8, shape=(10,))
In [2]: q[:] = arange(0,100,10)
memmap([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90], dtype=uint8)
```

new_file.dat




```
# Now, create a new memory mapped array (read only) with an offset into
# the previously created file.
In [3]: q=memmap('new_file.dat',mode='r', dtype=uint8, shape=6, offset=4)
In [4]: q
memmap([40, 50, 60, 70, 80, 90], dtype=uint8)
```

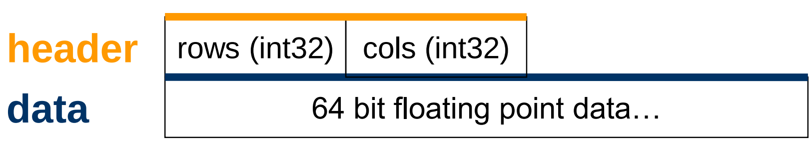
new_file.dat




```
# The number of bytes required by the array must be equal or less than
# the number of bytes available in the file.
In [3]: q=memmap('new_file.dat',mode='r', dtype=uint8, shape=7, offset=4)
ValueError: mmap length is greater than file size
```


Working with file headers

File Format:




```
# Create a dtype to represent the header.
header_dtype = dtype([('rows', int32), ('cols', int32)])

# Create a memory mapped array using this dtype.  Note the shape is empty.
header = memmap(file_name, mode='r', dtype=header_dtype, shape=())

# Read the row and column sizes from using this structured array.
rows = header['rows']
cols = header['cols']

# Create a memory map to the data segment, using rows, cols for shape
# information and the header size to determine the correct offset.
data = memmap(file_name, mode='r+', dtype=float64,
              shape=(rows, cols), offset=header_dtype.itemsize)
```

memory maps with ndarray

File Format:

header	rows (int32)	cols (int32)
data	64 bit floating point data…	

```
# mmap is a standard Python module for working with memory maps.
import mmap
import numpy

# Create a dtype to represent the header.
header_dtype = numpy.dtype([('rows', int32), ('cols', int32)])

# Open a file for read/write access in binary mode.
file = open(file_name,'r+b')

# Create a read-only memory map from the opened file with the
# correct size to read the header of the file.
mm = mmap.mmap(file.fileno(), header_dtype.itemsize,
               access=mmap.ACCESS_READ)
< continued >
```


memory maps with ndarray

File Format:

header	rows (int32)	cols (int32)
data	64 bit floating point data…	

```
# Create a new array using the ndarray constructor.
# The first argument is the shape, and we pass in the data type and the
# memory buffer to use (mm) as keyword arguments.
header = numpy.ndarray((), dtype=header_dtype, buffer=mm)
rows = header['rows']
cols = header['cols']

# Create a writable memory map to use for the data array.  The size of the
# memory map in bytes is the size of a float64 (8) * rows * columns.
mm = mmap.mmap(file.fileno(), 8*rows*cols, access=mmap.ACCESS_WRITE)

# Create our data array using this new memory map.  Start the arrays
# data at the memory location directly after the header using offset.
data = numpy.ndarray((rows, cols), dtype=float64, buffer=mm,
                     offset=header_dtype.itemsize)
```

ENTHOUGHT

Structured Arrays

char[12]	int64	float32
Name	**Time**	**Value**
MSFT_profit	10	6.20
GOOG_profit	12	-1.08
MSFT_profit	18	8.40
INTC_profit	25	-0.20
⋮	⋮	⋮
GOOG_profit	1000325	3.20
GOOG_profit	1000350	4.50
INTC_profit	1000385	-1.05
MSFT_profit	1000390	5.60

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

Example

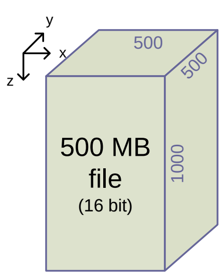
```
>>> import numpy as np
>>> fmt = np.dtype([('name', 'S12'),
                    ('time', np.int64),
                   :('value', np.float32)])
>>> vals = [('MSFT_profit', 10, 6.20),
            ('GOOG_profit', 12, -1.08),

            ('INTC_profit', 1000385, -1.05)
            ('MSFT_profit', 1000390, 5.60)]

>>> arr = np.array(vals, dtype=fmt)
# or
>>> arr = np.fromfile('db.dat', dtype=fmt)
# or
>>> arr = np.memmap('db.dat', dtype=fmt,
                    mode='c')
```

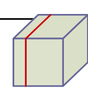
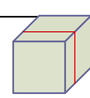
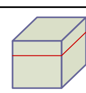
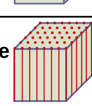
Disk

MSFT_profit	10	6.20	GOOG_profit	12	-1.08	• • •	INTC_profit	1000385	-1.05	MSFT_profit	1000390	5.60

ENTHOUGHT

Memmap Timings (3D arrays)

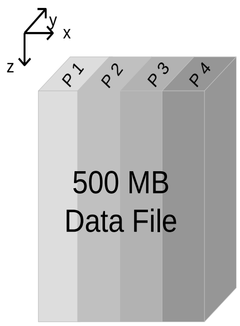


Operations (500x500x1000)		Linux		OS X	
		In Memory	**Memory Mapped**	**In Memory**	**Memory Mapped**
read		**2103 ms**	**11.0 ms**	**3505.00**	**27.00**
x slice		**1.8 ms**	**4.8 ms**	**1.80**	**8.30**
y slice		**2.8 ms**	**4.6 ms**	**4.40**	**7.40**
z slice		**9.2 ms**	**13.8 ms**	**10.40**	**18.70**
downsample 4x4		**0.02 ms**	**125 ms**	**0.02**	**198.70**

All times in millesconds (ms).

Linux: Ubuntu 4.10, Dell Precision 690, Dual Quad Core Zeon X5355 2.6 GHz, 8 GB Memory
OS X: OS X 10.5, MacBook Pro Laptop, 2.6 GHz Core Duo, 4 GB Memory

Parallel FFT On Memory Mapped File



Processors	Time (seconds)	Speed Up
1	11.75	1.0
2	6.06	1.9
4	3.36	3.5
8	2.50	4.7

```
from numpy import ceil
from ipython1.kernel import client
from geoio import vtio

# Execute an fft on a sub-section of a seismic cube.
code = \
"""
from numpy import fft
from geoio import vtio

seismic, params = vtio.read(file_name,rescale=False)
start, end = id*size, (id+1) * size
local_seismic = vtio.unclip(seismic[start:end,:,:])
spectrum = fft.fft(local_seismic,axis=-1)
"""

def equal_size_split(ary, cluster):
    # Return the number of rows each worker should work
    return int(ceil(float(len(ary))/len(cluster)))

# Run parallel code  on each of the remote processors
file_name = "500_500_1000.vt"
cluster = client.MultiEngineClient(('127.0.0.1',10105))
seismic, params = vtio.read(file_name,rescale=False)
cluster['size'] = equal_size_split(seismic, cluster)
cluster['file_name'] = file_name
cluster.execute(code)
```


Controlling Output Format

```
set_printoptions(precision=None, threshold=None,
                 edgeitems=None, linewidth=None,
                 suppress=None)
```

precision The number of digits of precision to use for floating point output. The default is 8.

threshold Array length where NumPy starts truncating the output and prints only the beginning and end of the array. The default is 1000.

edgeitems Number of array elements to print at beginning and end of array when threshold is exceeded. The default is 3.

linewidth Characters to print per line of output. The default is 75.

suppress Indicates whether NumPy suppresses printing small floating point values in scientific notation. The default is **False**.

ENTHOUGHT

Controlling Output Formats

PRECISION

```
>>> a = arange(1e6)
>>> a
array([  0.00000000e+00, 1.00000000e+00, 2.00000000e+00, ...,
         9.99997000e+05, 9.99998000e+05, 9.99999000e+05])
>>> set_printoptions(precision=3)
>>> a
array([  0.000e+00,   1.000e+00,   2.000e+00, ...,
         1.000e+06,   1.000e+06,   1.000e+06])
```

SUPPRESSING SMALL NUMBERS

```
>>> set_printoptions(precision=8)
>>> a = array((1, 2, 3, 1e-15))
>>> a
array([  1.00000000e+00,   2.00000000e+00,   3.00000000e+00,
         1.00000000e-15])
>>> set_printoptions(suppress=True)
>>> a
array([ 1.,   2.,   3.,   0.])
```


ENTHOUGHT

Controlling Error Handling

```
seterr(all=None, divide=None, over=None,
       under=None, invalid=None)
```

Set the error handling flags in ufunc operations on a per thread basis. Each of the keyword arguments can be set to ‘ignore’, ‘warn’, ‘print’, ‘log’, ‘raise’, or ‘call’.

all	All error types to the specified value
divide	Divide-by-zero errors
over	Overflow errors
under	Underflow errors
invalid	Invalid floating point errors

Controlling Error Handling

```
>>> a = array((1,2,3))
>>> a/0.
Warning: divide by zero encountered in divide
array([ 1.#INF0000e+000,  1.#INF0000e+000,  1.#INF0000e+000])

# Ignore division-by-zero.  Also, save old values so that
# we can restore them.
>>> old_err = seterr(divide='ignore')
>>> a/0.
array([ 1.#INF0000e+000,  1.#INF0000e+000,  1.#INF0000e+000])

# Restore original error handling mode.
>>> old_err
{'divide': 'print', 'invalid': 'print', 'over': 'print',
'under': 'ignore'}
>>> seterr(**old_err)
>>> a/0.
Warning: divide by zero encountered in divide
array([ 1.#INF0000e+000,  1.#INF0000e+000,  1.#INF0000e+000])
```