enthought ®

Introduction to Scientific Computing with Python

Eric Jones
eric@enthought.com

Enthought
www.enthought.com

Travis Oliphant
oliphant@ee.byu.edu

Brigham Young University
http://www.ee.byu.edu/

Topics

- Introduction to Python
- Numeric Computing
- SciPy
- Basic 2D Visualization

What Is Python?

ONE LINER

Python is an interpreted programming language that allows you to do almost anything possible with a compiled language (C/C++/Fortran) without requiring all the complexity.

PYTHON HIGHLIGHTS

- **Automatic garbage collection**
- **Dynamic typing**
- **Interpreted and interactive**
- **Object-oriented**
- **"Batteries Included"**
- **Free**
- **Portable**
- **Easy to Learn and Use**
- **Truly Modular**

enthought ®

BYU

Who is using Python?

NATIONAL SPACE TELESCOPE LABORATORY

Data processing and calibration for instruments on the Hubble Space Telescope.

INDUSTRIAL LIGHT AND MAGIC

Digital Animation

PAINT SHOP PRO 8

Scripting Engine for JASC PaintShop Pro 8 photo-editing software

CONOCOPHILLIPS

Oil exploration tool suite

LAWRENCE LIVERMORE NATIONAL LABORATORIES

Scripting and extending parallel physics codes. pyMPI is their doing.

WALT DISNEY

Digital animation development environment.

REDHAT

Anaconda, the Redhat Linux installer program, is written in Python.

ENTHOUGHT

Geophysics and Electromagnetics engine scripting, algorithm development, and visualization

Language Introduction

enthought ®

Interactive Calculator

```
# adding two values
>>> 1 + 1
2
# setting a variable
>>> a = 1
>>> a
1
# checking a variables type
>>> type(a)
<type 'int'>
# an arbitrarily long integer
>>> a = 1203405503201
>>> a
1203405503201L
>>> type(a)
<type 'long'>
```

```
# real numbers
>>> b = 1.2 + 3.1
>>> b
4.2999999999999998
>>> type(b)
<type 'float'>
# complex numbers
>>> c = 2+1.5j
>>> c
(2+1.5j)
```

The four numeric types in Python on 32-bit architectures are:

- **integer** (4 byte)
- **long integer** (any precision)
- **float** (8 byte like C's double)
- **complex** (16 byte)

The Numeric module, which we will see later, supports a larger number of numeric types.

Strings

CREATING STRINGS

```
# using double quotes
>>> s = "hello world"
>>> print s
hello world
# single quotes also work
>>> s = 'hello world'
>>> print s
hello world
```

STRING OPERATIONS

```
# concatenating two strings
>>> "hello " + "world"
'hello world'

# repeating a string
>>> "hello " * 3
'hello hello hello '
```

STRING LENGTH

```
>>> s = "12345"
>>> len(s)
5
```

FORMAT STRINGS

```
# the % operator allows you
# to supply values to a
# format string.  The format
# string follows
# C conventions.
>>> s = "some numbers:"
>>> x = 1.34
>>> y = 2
>>> s = "%s %f, %d" % (s,x,y)
>>> print s
some numbers: 1.34, 2
```

The string module


```
>>> import string
>>> s = “hello world”

# split space delimited words
>>> wrd_lst = string.split(s)
>>> print wrd_lst
[‘hello’, ‘world’]

# python2.2 and higher
>>> s.split()
[‘hello’, ‘world’]

# join words back together
>>> string.join(wrd_lst)
hello world

# python2.2 and higher
>>> ‘ ‘.join(wrd_lst)
hello world
```

```
# replacing text in a string
>>> string.replace(s,’world’ \
... ,’Mars’)
‘hello Mars’

# python2.2 and higher
>>> s.replace(’world’ ,’Mars’)
‘hello Mars’

# strip whitespace from string
>>> s = “\t   hello   \n”
>>> string.strip(s)
‘hello’

# python2.2 and higher
>>> s.strip()
‘hello’
```

Multi-line Strings


```
# triple quotes are used
# for mutli-line strings
>>> a = """hello
... world"""
>>> print a
hello
world

# multi-line strings using
# "\" to indicate
continuation
>>> a = "hello " \
...     "world"
>>> print a
hello world
```

```
# including the new line
>>> a = "hello\n" \
...     "world"
>>> print a
hello
world
```

List objects

LIST CREATION WITH BRACKETS

```
>>> l = [10,11,12,13,14]
>>> print l
[10, 11, 12, 13, 14]
```

CONCATENATING LIST

```
# simply use the + operator
>>> [10, 11] + [12,13]
[10, 11, 12, 13]
```

REPEATING ELEMENTS IN LISTS

```
# the multiply operator
# does the trick.
>>> [10, 11] * 3
[10, 11, 10, 11, 10, 11]
```

range(start, stop, step)

```
# the range method is helpful
# for creating a sequence
>>> range(5)
[0, 1, 2, 3, 4]

>>> range(2,7)
[2, 3, 4, 5, 6]

>>> range(2,7,2)
[2, 4, 6]
```

Indexing

RETREIVING AN ELEMENT

```
# list
# indices: 0   1   2   3   4
  >>>  l = [10,11,12,13,14]
  >>>  l[0]
  10
```

SETTING AN ELEMENT

```
>>> l[1] = 21
>>> print l
[10, 21, 12, 13, 14]
```

OUT OF BOUNDS

```
>>> l[10]
Traceback (innermost last):
File "<interactive input>",line 1,in ?
IndexError: list index out of range
```

NEGATIVE INDICES

```
# negative indices count
# backward from the end of
# the list.
#
# indices: -5 -4 -3 -2 -1
  >>>   l = [10,11,12,13,14]

  >>> l[-1]
  14
  >>> l[-2]
  13
```

The first element in an array has index=0 as in C. *Take note Fortran programmers!*

More on list objects

enthought ®

BYU

LIST CONTAINING MULTIPLE TYPES

```
# list containing integer,
# string, and another list.
>>> l = [10,'eleven',[12,13]]
>>> l[1]
'eleven'
>>> l[2]
[12, 13]

# use multiple indices to
# retrieve elements from
# nested lists.
>>> l[2][0]
12
```

LENGTH OF A LIST

```
>>> len(l)
3
```

DELETING OBJECT FROM LIST

```
# use the del keyword
>>> del l[2]
>>> l
[10,'eleven']
```

DOES THE LIST CONTAIN x ?

```
# use in or not in
>>> l = [10,11,12,13,14]
>>> 13 in l
1
>>> 13 not in l
0
```

Slicing

var[lower:upper]

Slices extract a portion of a sequence by specifying a lower and upper bound. The extracted elements start at lower and go up to, *but do not include*, the upper element. Mathematically the range is [lower,upper).

SLICING LISTS

```
# indices: 0   1   2   3   4
   >>> l = [10,11,12,13,14]
   # [10,11,12,13,14]
   >>> l[1:3]
   [11, 12]

# negative indices work also
>>> l[1:-2]
[11, 12]
>>> l[-4:3]
[11, 12]
```

OMITTING INDICES

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

# grab first three elements
>>> l[:3]
[10,11,12]
# grab last two elements
>>> l[-2:]
[13,14]
```

A few methods for list objects

enthought®

BYU

some_list.append(x)

Add the element x to the end of the list, `some_list`.

some_list.count(x)

Count the number of times x occurs in the list.

some_list.index(x)

Return the index of the first occurrence of x in the list.

some_list.remove(x)

Delete the first occurrence of x from the list.

some_list.reverse( )

Reverse the order of elements in the list.

some_list.sort(cmp)

By default, sort the elements in ascending order. If a compare function is given, use it to sort the list.

List methods in action

enthought ®

BYU

```
>>> l = [10,21,23,11,24]

# add an element to the list
>>> l.append(11)
>>> print l
[10,21,23,11,24,11]

# how many 11s are there?
>>> l.count(11)
2

# where does 11 first occur?
>>> l.index(11)
3
```

```
# remove the first 11
>>> l.remove(11)
>>> print l
[10,21,23,24,11]

# sort the list
>>> l.sort()
>>> print l
[10,11,21,23,24]

# reverse the list
>>> l.reverse()
>>> print l
[24,23,21,11,10]
```

Mutable vs. Immutable

MUTABLE OBJECTS

```
# Mutable objects, such as
# lists, can be changed
# in-place.

# insert new values into list
>>> l = [10,11,12,13,14]
>>> l[1:3] = [5,6]
>>> print l
[10, 5, 6, 13, 14]
```

The cStringIO module treats strings like a file buffer and allows insertions. It's useful when working with large strings or when speed is paramount.

IMMUTABLE OBJECTS

```
# Immutable objects, such as
# strings, cannot be changed
# in-place.

# try inserting values into
# a string
>>> s = 'abcde'
>>> s[1:3] = 'xy'
Traceback (innermost last):
File "<interactive input>",line 1,in ?
TypeError: object doesn't support
          slice assignment

# here's how to do it
>>> s = s[:1] + 'xy' + s[3:]
>>> print s
'axyde'
```

Dictionaries

enthought ®

BYU

Dictionaries store *key*/*value* pairs. Indexing a dictionary by a *key* returns the *value* associated with it.

DICTIONARY EXAMPLE

```
# create an empty dictionary using curly brackets
>>> record = {}
>>> record['first'] = 'Jmes'
>>> record['last'] = 'Maxwell'
>>> record['born'] = 1831
>>> print record
{'first': 'Jmes', 'born': 1831, 'last': 'Maxwell'}
# create another dictionary with initial entries
>>> new_record = {'first': 'James', 'middle':'Clerk'}
# now update the first dictionary with values from the new one
>>> record.update(new_record)
>>> print record
{'first': 'James', 'middle': 'Clerk', 'last':'Maxwell', 'born':
1831}
```

A few dictionary methods

enthought ®

BYU

some_dict.clear( )

Remove all key/value pairs from the dictionary, `some_dict`.

some_dict.copy( )

Create a copy of the dictionary

some_dict.has_key(x)

Test whether the dictionary contains the key `x`.

some_dict.keys( )

Return a list of all the keys in the dictionary.

some_dict.values( )

Return a list of all the values in the dictionary.

some_dict.items( )

Return a list of all the key/value pairs in the dictionary.

Dictionary methods in action

enthought ®

BYU

```
>>> d = {‘cows’: 1,’dogs’:5,
...      ‘cats’: 3}

# create a copy.
>>> dd = d.copy()
>>> print dd
{'dogs':5,'cats':3,'cows': 1}

# test for chickens.
>>> d.has_key(‘chickens’)
0

# get a list of all keys
>>> d.keys()
[‘cats’,’dogs’,’cows’]
```

```
# get a list of all values
>>> d.values()
[3, 5, 1]

# return the key/value pairs
>>> d.items()
[('cats', 3), ('dogs', 5),
 ('cows', 1)]

# clear the dictionary
>>> d.clear()
>>> print d
{}
```

enthought ®

BYU

Tuples

Tuples are a sequence of objects just like lists. Unlike lists, tuples are immutable objects. While there are some functions and statements that require tuples, they are rare. A good rule of thumb is to use lists whenever you need a generic sequence.

TUPLE EXAMPLE

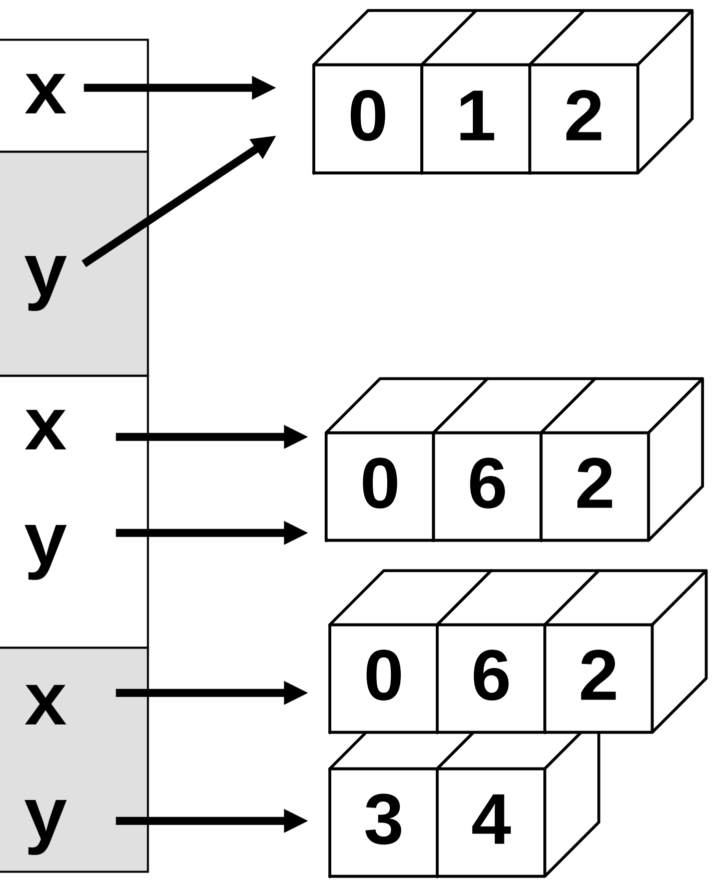
```
# tuples are built from a comma separated list enclosed by ( )
>>> t = (1,’two’)
>>> print t
(1,‘two’)
>>> t[0]
1
# assignments to tuples fail
>>> t[0] = 2
Traceback (innermost last):
File "<interactive input>", line 1, in ?
TypeError: object doesn't support item assignment
```

Assignment

enthought ®

Assignment creates object references.

```
>>> x = [0, 1, 2]

# y = x cause x and y to point
# at the same list
>>> y = x

# changes to y also change x
>>> y[1] = 6
>>> print x
[0, 6, 2]

# re-assigning y to a new list
# decouples the two lists
>>> y = [3, 4]
```



Multiple assignments


```
# creating a tuple without ()
>>> d = 1,2,3
>>> d
(1, 2, 3)
```

```
# multiple assignments
>>> a,b,c = 1,2,3
>>> print b
2
```

```
# multiple assignments from a
# tuple
>>> a,b,c = d
>>> print b
2
```

```
# also works for lists
>>> a,b,c = [1,2,3]
>>> print b
2
```

If statements

`if/elif/else` provide conditional execution of code blocks.

IF STATEMENT FORMAT

```
if <condition>:
     <statements>
elif <condition>:
     <statements>
else:
     <statements>
```

IF EXAMPLE

```
# a simple if statement
>>> x = 10
>>> if x > 0:
...     print 1
... elif x == 0:
...     print 0
... else:
...     print -1
... < hit return >
1
```

Test Values

- True means any non-zero number or non-empty object
- False means not true: zero, empty object, or **None**

EMPTY OBJECTS

```
# empty objects evaluate false
>>> x = []
>>> if x:
...     print 1
... else:
...     print 0
... < hit return >
0
```

For loops

For loops iterate over a sequence of objects.

```
for <loop_var> in <sequence>:
     <statements>
```

TYPICAL SCENARIO

```
>>> for i in range(5):
...     print i,
... < hit return >
0 1 2 3 4
```

LOOPING OVER A STRING

```
>>> for i in 'abcde':
...     print i,
... < hit return >
a b c d e
```

LOOPING OVER A LIST

```
>>> l=['dogs','cats','bears']
>>> accum = ''
>>> for item in l:
...     accum = accum + item
...     accum = accum + ' '
... < hit return >
>>> print accum
dogs cats bears
```

While loops

While loops iterate until a condition is met.

```
while <condition>:
    <statements>
```

WHILE LOOP

```
# the condition tested is
# whether lst is empty.
>>> lst = range(3)
>>> while lst:
...     print lst
...     lst = lst[1:]
... < hit return >
[0, 1, 2]
[1, 2]
[2]
```

BREAKING OUT OF A LOOP

```
# breaking from an infinite
# loop.
>>> i = 0
>>> while 1:
...     if i < 3:
...         print i,
...     else:
...         break
...     i = i + 1
... < hit return >
0 1 2
```

Anatomy of a function

The keyword **def** indicates the start of a function.

Function arguments are listed separated by commas. They are passed by *assignment*. More on this later.

```
def add(arg0, arg1):
    a = arg0 + arg1
    return a
```

Indentation is used to indicate the contents of the function. It is *not* optional, but a part of the syntax.

A colon (**:**) terminates the function definition.

An optional return statement specifies the value returned from the function. If return is omitted, the function returns the special value **None**.

Our new function in action

enthought ®

```
# We'll create our function
# on the fly in the
# interpreter.
>>> def add(x,y):
...     a = x + y
...     return a

# test it out with numbers
>>> x = 2
>>> y = 3
>>> add(x,y)
5
```

```
# how about strings?
>>> x = 'foo'
>>> y = 'bar'
>>> add(x,y)
'foobar'

# functions can be assigned
# to variables
>>> func = add
>>> func(x,y)
'foobar'
```

```
# how about numbers and strings?
>>> add('abc',1)
Traceback (innermost last):
File "<interactive input>", line 1, in ?
File "<interactive input>", line 2, in add
TypeError: cannot add type "int" to string
```

Modules

enthought ®

BYU

EX1.PY

```python
# ex1.py

PI = 3.1416

def sum(lst):
    tot = lst[0]
    for value in lst[1:]:
        tot = tot + value
    return tot

l = [0,1,2,3]
print sum(l), PI
```

FROM SHELL

```
[ej@bull ej]$ python ex1.py
6, 3.1416
```

FROM INTERPRETER

```python
# load and execute the module
>>> import ex1
6, 3.1416
# get/set a module variable.
>>> ex1.PI
3.1415999999999999
>>> ex1.PI = 3.14159
>>> ex1.PI
3.1415899999999999
# call a module variable.
>>> t = [2,3,4]
>>> ex1.sum(t)
9
```

Modules *cont.*

enthought ®

BYU

INTERPRETER

```
# load and execute the module
>>> import ex1
6, 3.1416
< edit file >
# import module again
>>> import ex1
# nothing happens!!!


# use reload to force a
# previously imported library
# to be reloaded.
>>> reload(ex1)
10, 3.14159
```

EDITED EX1.PY

```
# ex1.py version 2

PI = 3.14159

def sum(lst):
    tot = 0
    for value in lst:
        tot = tot + value
    return tot

l = [0,1,2,3,4]
print sum(l), PI
```

Modules *cont.* 2

enthought ®

BYU

Modules can be executable scripts or libraries or both.

EX2.PY

```
" An example module "

PI = 3.1416

def sum(lst):
    """ Sum the values in a
        list.
    """
    tot = 0
    for value in lst:
        tot = tot + value
    return tot
```

EX2.PY CONTINUED

```
def add(x,y):
    " Add two values."
    a = x + y
    return a

def test():
    l = [0,1,2,3]
    assert( sum(l) == 6)
    print 'test passed'

# this code runs only if this
# module is the main program
if __name__ == '__main__':
    test()
```

Setting up PYTHONPATH

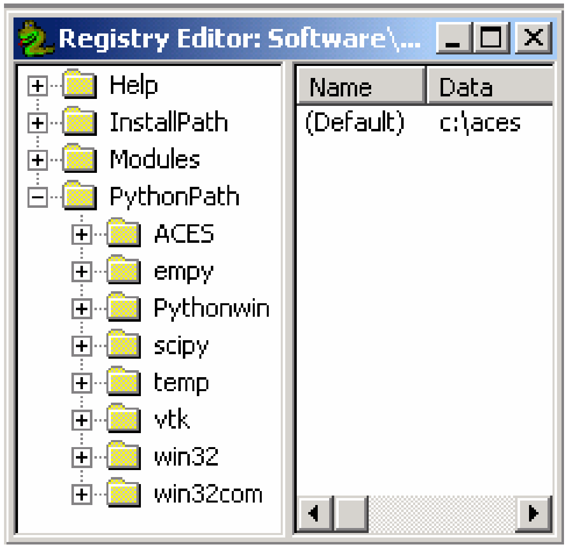
enthought ®

BYU

PYTHONPATH is an environment variable (or set of registry entries on Windows) that lists the directories Python searches for modules.

WINDOWS

The easiest way to set the search paths is using PythonWin's *Tools->Edit Python Path* menu item. Restart PythonWin after changing to insure changes take affect.



UNIX -- .cshrc

```
!! note: the following should !!
!! all be on one line         !!

setenv PYTHONPATH
    $PYTHONPATH:$HOME/aces
```

UNIX -- .bashrc

```
PYTHONPATH=$PYTHONPATH:$HOME/aces
export PYTHONPATH
```

Classes

SIMPLE PARTICLE CLASS

```
>>> class particle:
...     # Constructor method
...     def __init__(self,mass, velocity):
...         # assign attribute values of new object
...         self.mass = mass
...         self.velocity = velocity
...     # method for calculating object momentum
...     def momentum(self):
...         return self.mass * self.velocity
...     # a “magic” method defines object’s string representation
...     def __repr__(self):
...         msg = "(m:%2.1f, v:%2.1f)" % (self.mass,self.velocity)
...         return msg
```

EXAMPLE

```
>>> a = particle(3.2,4.1)
>>> a
(m:3.2, v:4.1)
>>> a.momentum()
13.119999999999999
```

Reading files

enthought ®

BYU

FILE INPUT EXAMPLE

```
>>> results = []
>>> f = open('c:\\rcs.txt','r')

# read lines and discard header
>>> lines = f.readlines()[1:]
>>> f.close()

>>> for l in lines:
...     # split line into fields
...     fields = line.split()
...     # convert text to numbers
...     freq = float(fields[0])
...     vv = float(fields[1])
...     hh = float(fields[2])
...     # group & append to results
...     all = [freq,vv,hh]
...     results.append(all)
... < hit return >
```

PRINTING THE RESULTS

```
>>> for i in results: print i
[100.0, -20.30…, -31.20…]
[200.0, -22.70…, -33.60…]
```

EXAMPLE FILE: RCS.TXT

```
#freq (MHz)  vv (dB)  hh (dB)
  100        -20.3     -31.2
  200        -22.7     -33.6
```

enthought ®

More compact version

ITERATING ON A FILE AND LIST COMPREHENSIONS

```
>>> results = []
>>> f = open('c:\\rcs.txt','r')
>>> f.readline()
'#freq (MHz)  vv (dB)  hh (dB)\n'
>>> for l in f:
...    all = [float(val) for val in l.split()]
...    results.append(all)
... < hit return >
>>> for i in results:
...    print i
... < hit return >
```

EXAMPLE FILE: RCS.TXT

```
#freq (MHz)  vv (dB)  hh (dB)
  100        -20.3    -31.2
  200        -22.7    -33.6
```

Same thing, one line

enthought ®

BYU

OBFUSCATED PYTHON CONTEST…

```
>>> print [[float(val) for val in l.split()] for
...         l in open("c:\\temp\\rcs.txt","r")
...         if l[0] !="#"]
```

EXAMPLE FILE: RCS.TXT

```
#freq (MHz)   vv (dB)   hh (dB)
  100         -20.3     -31.2
  200         -22.7     -33.6
```

Pickling and Shelves

enthought ®

BYU

Pickling is Python's term for *persistence*. Pickling can write arbitrarily complex objects to a file. The object can be resurrected from the file at a later time for use in a program.

```
>>> import shelve
>>> f = shelve.open('c:/temp/pickle','w')
>>> import ex_material
>>> epoxy_gls = ex_material.constant_material(4.8,1)
>>> f['epoxy_glass'] = epoxy_gls
>>> f.close()
< kill interpreter and restart! >
>>> import shelve
>>> f = shelve.open('c:/temp/pickle','r')
>>> epoxy_glass = f['epoxy_glass']
>>> epoxy_glass.eps(100e6)
4.249e-11
```

Exception Handling

enthought ®

BYU

ERROR ON LOG OF ZERO

```
import math
>>> math.log10(10.)
1.
>>> math.log10(0.)
Traceback (innermost last):
OverflowError: math range error
```

CATCHING ERROR AND CONTINUING

```
>>> a = 0.
>>> try:
...     r = math.log10(a)
... except OverflowError:
...     print 'Warning: overflow occurred. Value set to 0.'
...     # set value to 0. and continue
...     r = 0.
Warning: overflow occurred. Value set to 0.
>>> print r
0.0
```

Sorting

enthought ®

BYU

THE CMP METHOD

```
# The builtin cmp(x,y)
# function compares two
# elements and returns
# -1, 0, 1
# x <  y --> -1
# x == y -->  0
# x >  y -->  1
>>> cmp(0,1)
-1

# By default, sorting uses
# the builtin cmp() method
>>> x = [1,4,2,3,0]
>>> x.sort()
>>> x
[0, 1, 2, 3, 4]
```

CUSTOM CMP METHODS

```
# define a custom sorting
# function to reverse the
# sort ordering
>>> def descending(x,y):
...     return -cmp(x,y)

# Try it out
>>> x.sort(descending)
>>> x
[4, 3, 2, 1, 0]
```

Sorting

SORTING CLASS INSTANCES

```
# Comparison functions for a variety of particle values
>>> def by_mass(x,y):
...     return cmp(x.mass,y.mass)
>>> def by_velocity(x,y):
...     return cmp(x.velocity,y.velocity)
>>> def by_momentum(x,y):
...     return cmp(x.momentum(),y.momentum())

# Sorting particles in a list by their various properties
>>> x = [particle(1.2,3.4),particle(2.1,2.3),particle(4.6,.7)]
>>> x.sort(by_mass)
>>> x
[(m:1.2, v:3.4), (m:2.1, v:2.3), (m:4.6, v:0.7)]
>>> x.sort(by_velocity)
>>> x
[(m:4.6, v:0.7), (m:2.1, v:2.3), (m:1.2, v:3.4)]
>>> x.sort(by_momentum)
>>> x
[(m:4.6, v:0.7), (m:1.2, v:3.4), (m:2.1, v:2.3)]
```

Numeric

Numeric

- Offers Matlab-ish capabilities within Python
- Download Site
  - http://sourceforge.net/projects/numpy/
- Developers (initial coding by Jim Hugunin)
  - Paul Dubouis
  - Travis Oliphant
  - Konrad Hinsen
  - Many more…

Numarray (nearing stable) is optimized for large arrays.

Numeric is more stable and is faster for operations on many small arrays.

Getting Started

IMPORT NUMERIC

```
>>> from Numeric import *
>>> import Numeric
>>> Numeric.__version__
'23.1'
          or
>>> from scipy import *
```

IMPORT PLOTTING TOOLS

```
>>> import gui_thread
>>> gui_thread.start()
>>> from scipy import plt
          or
>>> from scipy import xplt
          or
>>> from scipy import gplt
```

`gui_thread` starts wxPython in a second thread. Plots displayed within the second thread do not suspend the command line interpreter.

`plt` is wxPython based.

Compatible with: PythonWin,
- wxPython apps, Windows
- Command Line Python, Linux
- Command Line Python

`xplt` works well to produce 2-D graphs --- many features.

`gplt` wraps gnuplot – allows surface and 3-d plots.

Array Operations

SIMPLE ARRAY MATH

```
>>> a = array([1,2,3,4])
>>> b = array([2,3,4,5])
>>> a + b
array([3, 5, 7, 9])
```

Numeric defines the following constants:

```
pi = 3.14159265359
e  = 2.71828182846
```

MATH FUNCTIONS

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

# multiply entire array by
# scalar value
>>> a = (2*pi)/10.
>>> a
0.628318530718
>>> a*x
array([ 0.,0.628,…,6.283])

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

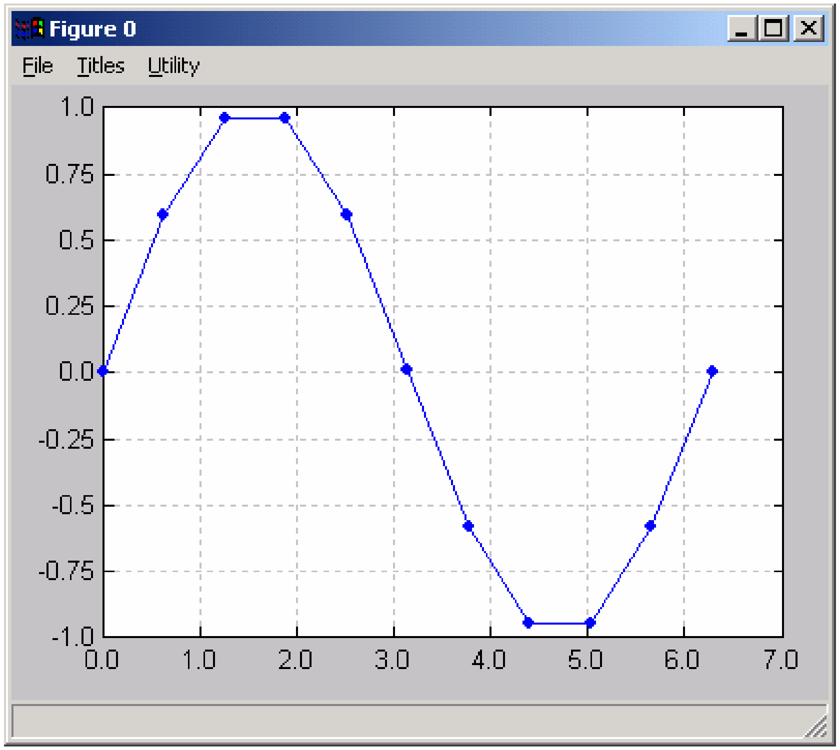
Plotting Arrays

enthought ®

BYU

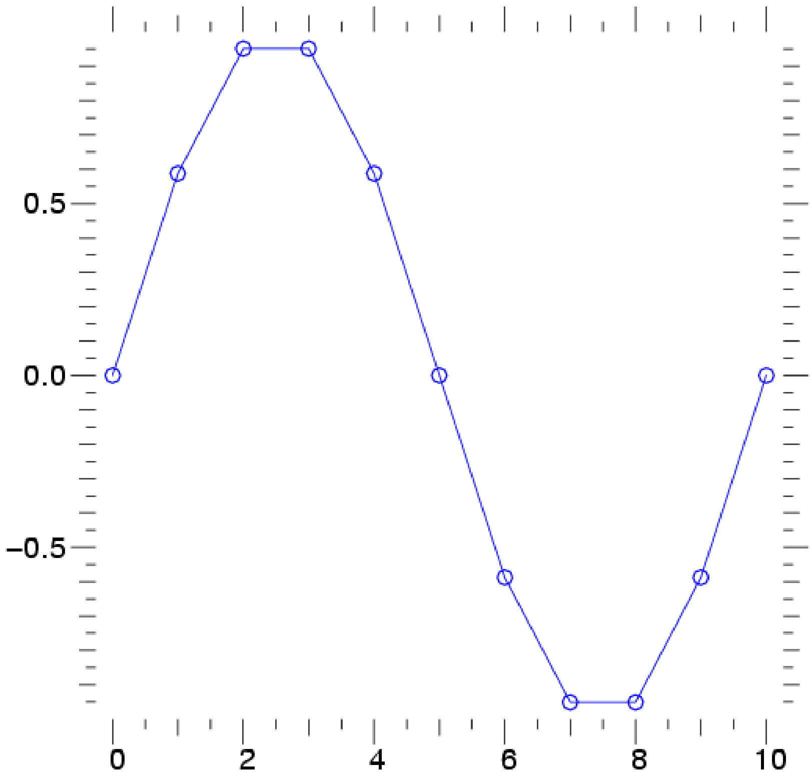
SCATTER PLOTS

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



SCATTER PLOTS

```
>>> xplt.plot(x,y,x,y,'bo')
```



enthought ®

BYU

Plotting Images

IMAGE PLOTS

```
>>> plt.image(lena())
```

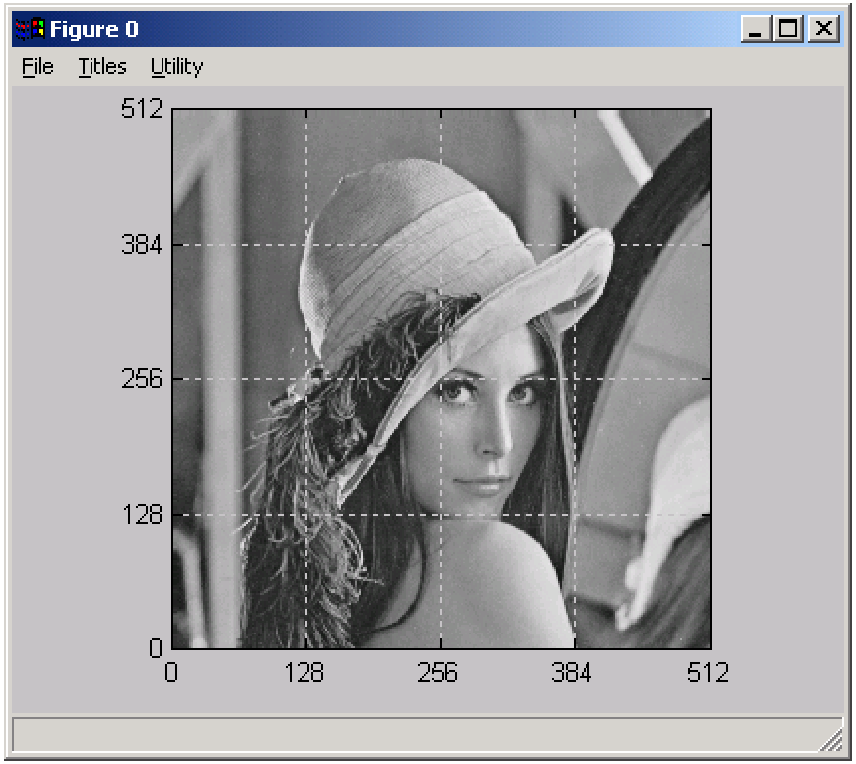
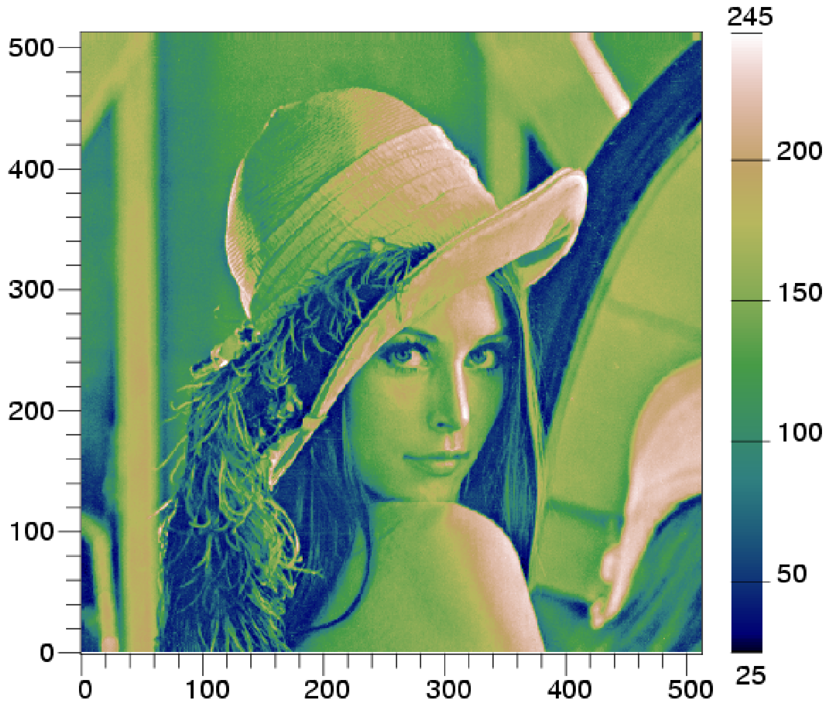


IMAGE PLOTS

```
>>> img = lena()[::-1]
>>> xplt.imagesc(img)
```



Introducing Numeric Arrays

enthought ®

BYU

SIMPLE ARRAY CREATION

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

CHECKING THE TYPE

```
>>> type(a)
<type 'array'>
```

NUMERIC TYPE OF ELEMENTS

```
>>> a.typecode()
'l'   # 'l' = Int
```

BYTES IN AN ARRAY ELEMENT

```
>>> a.itemsize()
4
```

ARRAY SHAPE

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

CONVERT TO PYTHON LIST

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

ARRAY INDEXING

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

Multi-Dimensional Arrays

enthought ®

BYU

MULTI-DIMENSIONAL ARRAYS

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

(ROWS,COLUMNS)

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

GET/SET ELEMENTS

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

column
row

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

ADDRESS FIRST ROW USING SINGLE INDEX

```
>>> a[1]
array([10, 11, 12, 13])
```

FLATTEN TO 1D ARRAY

```
>>> a.flat
array(0,1,2,3,10,11,12,-1)
>>> ravel(a)
array(0,1,2,3,10,11,12,-1)
```

A.FLAT AND RAVEL() REFERENCE ORIGINAL MEMORY

```
>>> a.flat[5] = -2
>>> a
array([[ 0, 1, 2, 3],
       [10,-2,12,-1]])
```

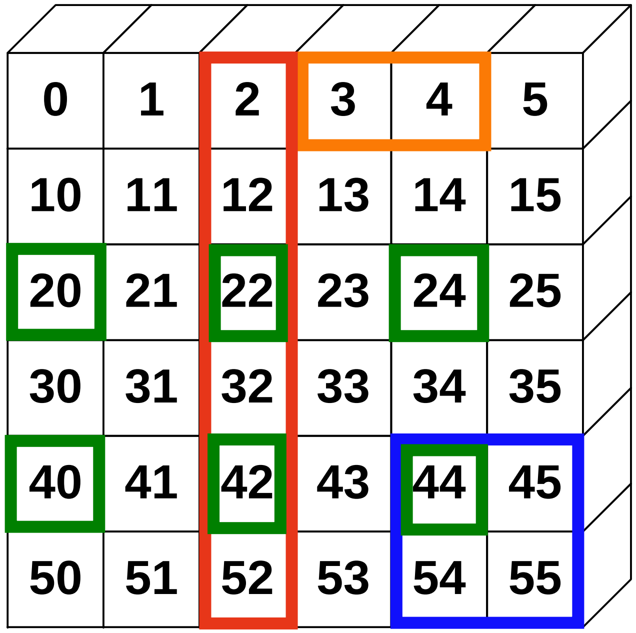
Array Slicing

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

enthought ®

BYU

Slices are references to memory in original array. Changing values in a slice also changes the original array.

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

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

# changing b changed a!
>>> a
array([ 1,  2, 10])
```

Array Constructor

```
array(sequence, typecode=None, copy=1, savespace=0)
```

`sequence` - any type of Python sequence. Nested list create multi-dimensional arrays.

`typecode` - character (string). Specifies the numerical type of the array. If it is None, the constructor makes its best guess at the numeric type.

`copy` - if **`copy=0`** and sequence is an array object, the returned array is a reference that data. Otherwise, a copy of the data in **`sequence`** is made.

`savespace` - Forces Numeric to use the smallest possible numeric type for the array. Also, it prevents upcasting to a different type during math operations with scalars. (see coercion section for more details)

Array Constructor Examples

enthought ®

BYU

FLOATING POINT ARRAYS DEFAULT TO DOUBLE PRECISION

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

notice decimal

USE TYPECODE TO REDUCE PRECISION

```
>>> a = array([0,1.,2,3],
...            typecode=Float32)
>>> a.typecode()
'f'
>>> len(a.flat)*a.itemsize()
16
```

BYTES FOR MAIN ARRAY STORAGE

```
# flat assures that
# multidimensional arrays
# work
>>>len(a.flat)*a.itemsize()
32
```

ARRAYS REFERENCING SAME DATA

```
>>> a = array((1,2,3,4))
>>> b = array(a,copy=0)
>>> b[1] = 10
>>> a
array([ 1, 10,  3,  4])
```

enthought ®

32-bit Typecodes

Character	Bits (Bytes)	Identifier
D	128 (16)	**Complex**, Complex64
F	64 (8)	Complex0, Complex8, Complex16, Complex32
d	64 (8)	**Float**, Float64
f	32 (4)	Float0, Float8, Float16, Float32
l	32 (4)	**Int**
i	32 (4)	Int32
s	16 (2)	Int16
1 (one)	8 (1)	Int8
u	32 (4)	UnsignedInt32
w	16 (2)	UnsignedInt16
b	8 (1)	UnsignedInt8
O	4 (1)	PyObject

Highlighted typecodes correspond to Python's standard Numeric types.

Array Creation Functions

`arange(start,stop=None,step=1,typecode=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`. When not specified, `typecode` is derived from the `start`, `stop`, and `step` values.

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

`ones(shape,typecode=None,savespace=0)`
`zeros(shape,typecode=None,savespace=0)`

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

```
>>> ones((2,3),typecode=Float32)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]],'f')
```

Array Creation Functions (cont.)

identity(n,typecode='l')

Generates an **n** by **n** identity matrix with **typecode = Int**.

```
>>> identity(4)
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
>>> identity(4,'f')
array([[ 1.,  0.,  0.,  0.],
           [ 0.,  1.,  0.,  0.],
           [ 0.,  0.,  1.,  0.],
           [ 0.,  0.,  0.,  1.]],'f')
```

Gotchas!

enthought ®

FORGETTING EXTRA () IN `array`

A common mistake is calling **`array`** with multiple arguments instead of a single sequence when creating arrays.

GOTCHA!

```
>>> a = array(0,1,2,3)
TypeError: …
```

REMEDY

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

WRONG ARRAY TYPE

`arange`, **`zeros`**, **`ones`**, and **`identity`** return **`Int`** arrays by default. This can cause unexpected behavior when setting values or during arithmetic.

GOTCHA!

```
>>> a = zeros((2,2))
>>> a[0,0] = 3.2
>>> a
array([[3, 0],[0, 0]])
```

REMEDY

```
>>> a = zeros((2,2),Float)
>>> a[0,0] = 3.2
>>> a
array([[ 3.2,0.],[0.,0.]])
```

Mathematic 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

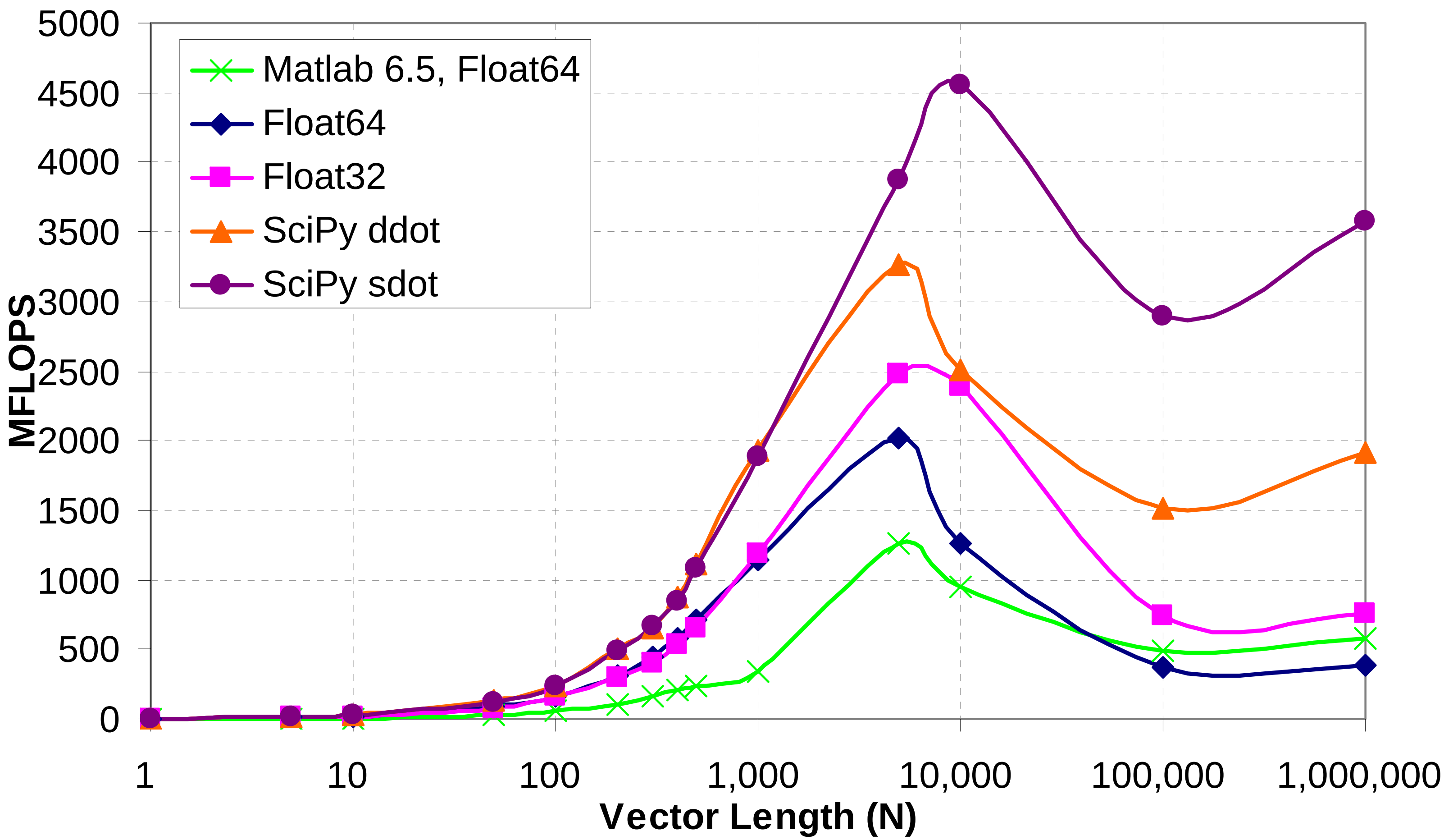
```
>>> add(a,b)
array([4, 6])
```

IN PLACE OPERATION

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

Vector Multiply Speed

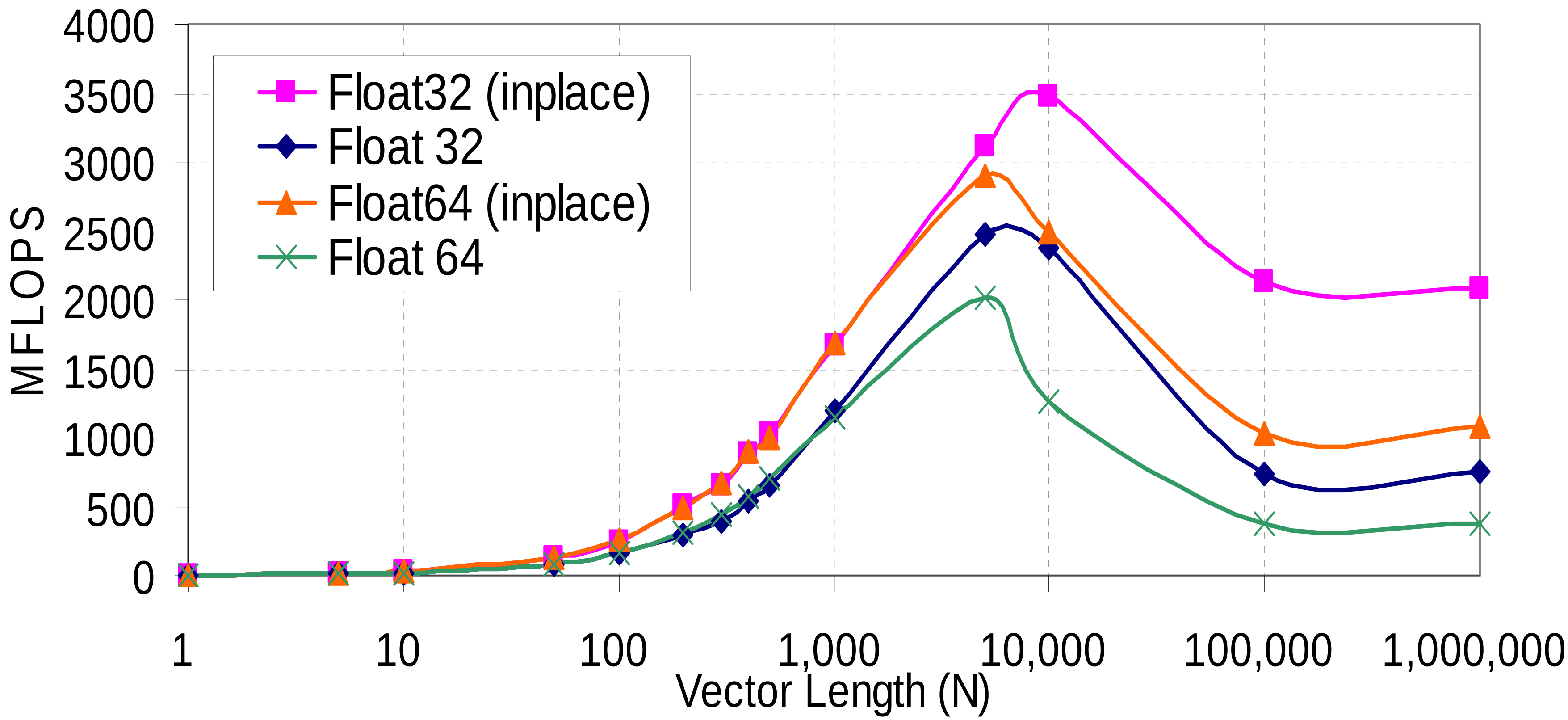
enthought ®



2.6 Ghz, Mandrake Linux 9.1, Python 2.3, Numeric 23.1, SciPy 0.2.0, gcc 3.2.2

Standard vs. "In Place" Multiply

enthought ®



2.6 Ghz, Mandrake Linux 9.1, Python 2.3, Numeric 23.1, SciPy 0.2.0, gcc 3.2.2

Your mileage can vary.

Numeric and SciPy Differences

enthought ®

BYU

NUMERIC

Numeric issues errors in most situations where `Inf` or `NaN`s are generated.

```
>>> from Numeric import *
>>>array((-1.,0,1))/array(0.)
OverflowError: math range error
>>>array((-1.,1))/ array(0.)
array([-1.#INF0e+0, 1.#INF0e+0])

>>> log(array((1,0.)))
OverflowError: math range error
```

SCIPY

SciPy carries the `Inf` and `NaN` values through the calculations. It also calculates complex values when appropriate.

```
>>> from scipy import *
>>> array((-1,0.,1.))/0.
array([-1.#INF,-1.#IND,1.#INF])

>>> log(array((1,0.,-1.)))
array([0.0+0.0j,
       -1.#INF0+0.0j,
       0.0+3.14159265j])
```

Comparison and Logical Operators

equal	(==)	not_equal	(!=)	greater	(>)
greater_equal	(>=)	less	(<)	less_equal	(<=)
logical_and	(and)	logical_or	(or)	logical_xor	
logical_not	(not)				

2D EXAMPLE

```
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[1, 1, 0, 1],
       [0, 1, 1, 1]])
# functional equivalent
>>> equal(a,b)
array([[1, 1, 0, 1],
       [0, 1, 1, 1]])
```

Bitwise Operators

enthought ®

BYU

```
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_and(a,b)
array([ 17,  34,  68, 136])

# bit inversion
>>> a = array((1,2,3,4),UnsignedInt8)
>>> invert(a)
array([254, 253, 252, 251],'b')

# surprising type conversion
>>> left_shift(a,3)
array([ 8, 16, 24, 32],'i')
```

Changed from UnsignedInt8 to Int32

Trig and Other Functions

enthought®

BYU

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)
```

hypot(x,y)

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

Universal Function Methods

The mathematic, comparative, logical, and bitwise operators 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 the 1d array `a` reducing it to a single value. Using `add` as an example:

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

ADD EXAMPLE

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

STRING LIST EXAMPLE

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

LOGICAL OP EXAMPLES

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

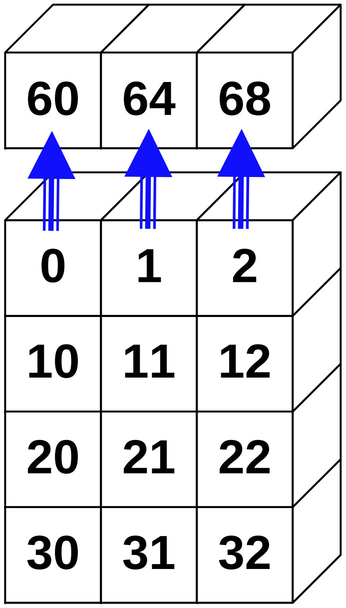
op.reduce()

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.

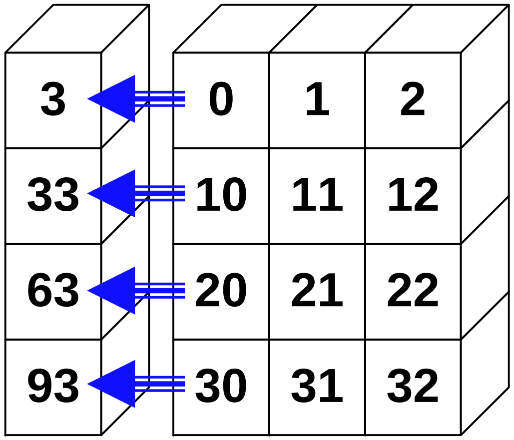
SUM COLUMNS BY DEFAULT

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



SUMMING UP EACH ROWS

```
>>> 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`.

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

ADD EXAMPLE

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

STRING LIST EXAMPLE

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

LOGICAL OP EXAMPLES

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

op.reduceat()

enthought®

BYU

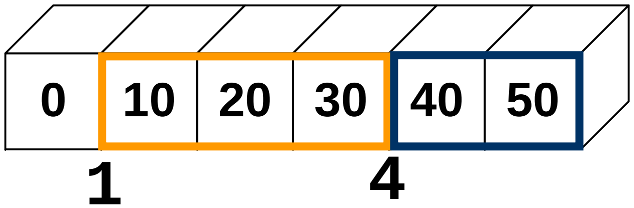
`op.reduceat(a,indices)` applies `op` to ranges in the 1d array `a` defined by the values in `indices`. The resulting array has the same length as `indices`.

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

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

EXAMPLE

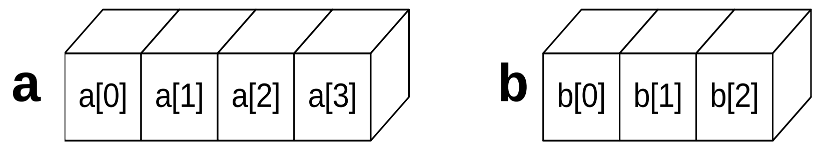
```
>>> a = array([0,10,20,30,
...            40,50])
>>> indices = array([1,4])
>>> add.reduceat(a,indices)
array([60, 90])
```



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

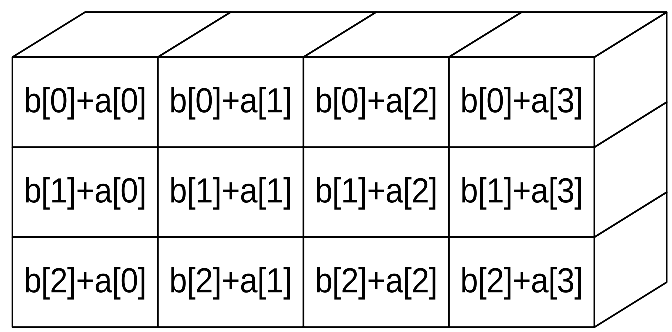
op.outer()

`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)
```



Type Casting

UPCASTING

`asarray()` only allows upcasting to higher precision

```
>>> a = array((1.2, -3),
...           typecode=Float32)
>>> a
array([ 1.20000005,-3.],'f')
# upcast
>>> asarray(a,
...          typecode=Float64)
array([ 1.20000005,-3.])

# failed downcast
>>> asarray(a,
...     typecode=UnsignedInt8)
TypeError: Array can not be
safely cast to required type
```

DOWNCASTING

`astype()` allows up or down casting, but may lose precision or result in unexpected conversions

```
>>> a = array((1.2,-3))
>>> a.astype(Float32)
array([ 1.20000005, -3.],'f')
>>> a.astype(UnsignedInt8)
array([  1, 253],'b')
```

Type Casting Gotchas!

enthought ®

BYU

PROBLEM

Silent upcasting converts a single precision array to double precision when operating with Python scalars.

```
>>> a = array([1,2,3,4,5],
... typecode=Float32)
>>> a.typecode()
‘f’
>>> b = a * 2.
>>> b.typecode()
‘d’
```

REMEDY 1

Create an array from the scalar and set its type correctly. (kinda ugly)

```
>>> two = array(2.,Float32)
>>> b = a * two
>>> b.typecode()
'f'
```

REMEDY 2

Set the array type to savespace=1. This prevents silent upcasting.

```
>>> a = array([1,2,3,4,5],
...         typecode = Float32,
...         savespace=1)
>>> b = a * 2.
>>> b.typecode()
‘f’
```

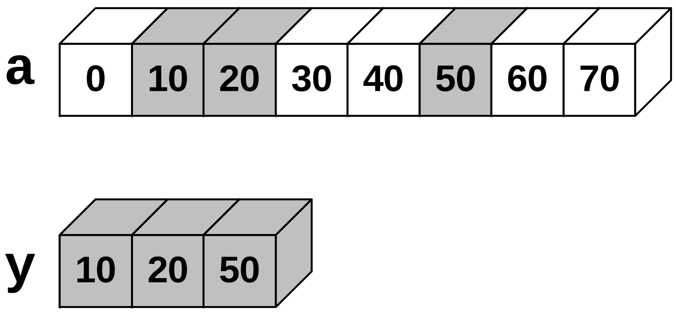
Array Functions – `take()`

`take(a,indices,axis=0)`

Create a new array containing slices from `a`. `indices` is an array specifying which slices are taken and `axis` the slicing axis. The new array contains copies of the data from `a`.

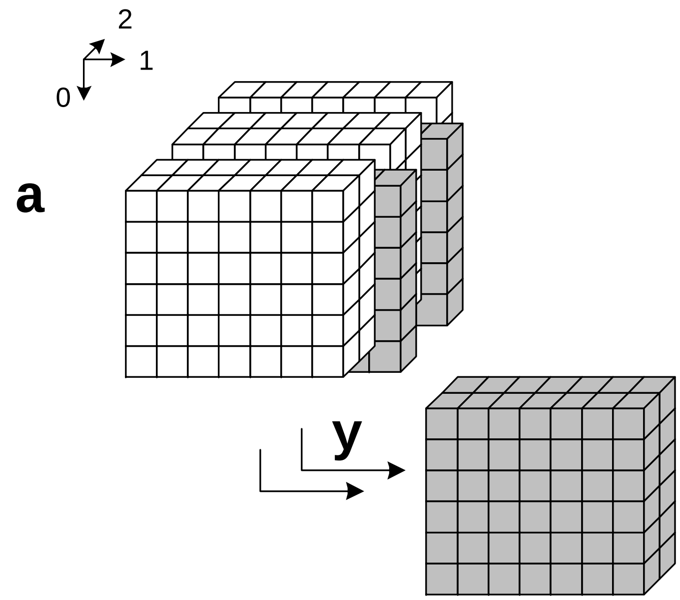
ONE DIMENSIONAL

```
>>> a = arange(0,80,10)
>>> y = take(a,[1,2,-3])
>>> print y
[10 20 50]
```



MULTIDIMENSIONAL

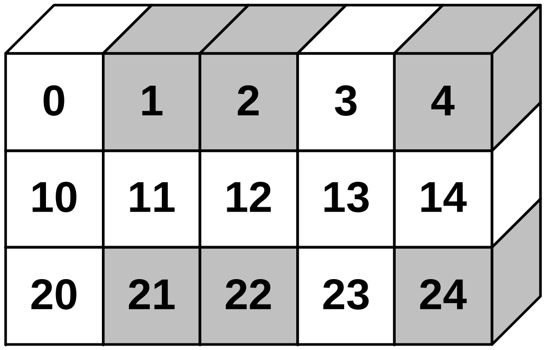
```
>>> y = take(a,[2,-2], 2)
```



Matlab vs. take( )

enthought ®

BYU



MATLAB ALLOWS ARRAYS AS INDICES

```
a =
     0   1   2   3   4
    10  11  12  13  14
    20  21  22  23  24
>>> a([1,3],[2,3,5])
ans =
     1   2   4
    21  22  24
```

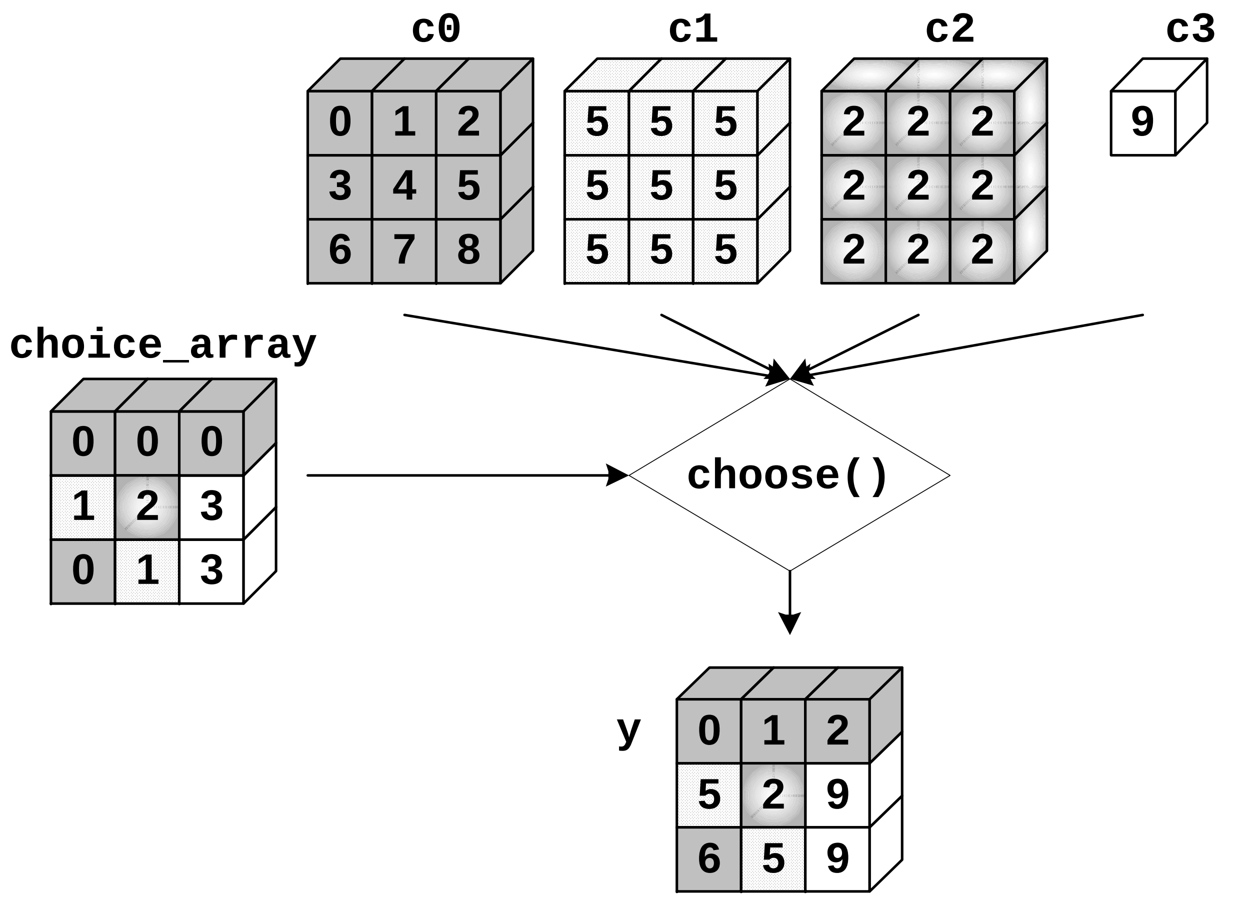
EQUIVALENT IN PYTHON

```
>>> a
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
>>> take(take(a,[0,2]),
...       [1,2,4],1)
array([[ 1,  2,  4],
       [21, 22, 24]])
```

Array Functions – choose( )

enthought ®

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



Example - choose( )

CLIP LOWER VALUES TO 10

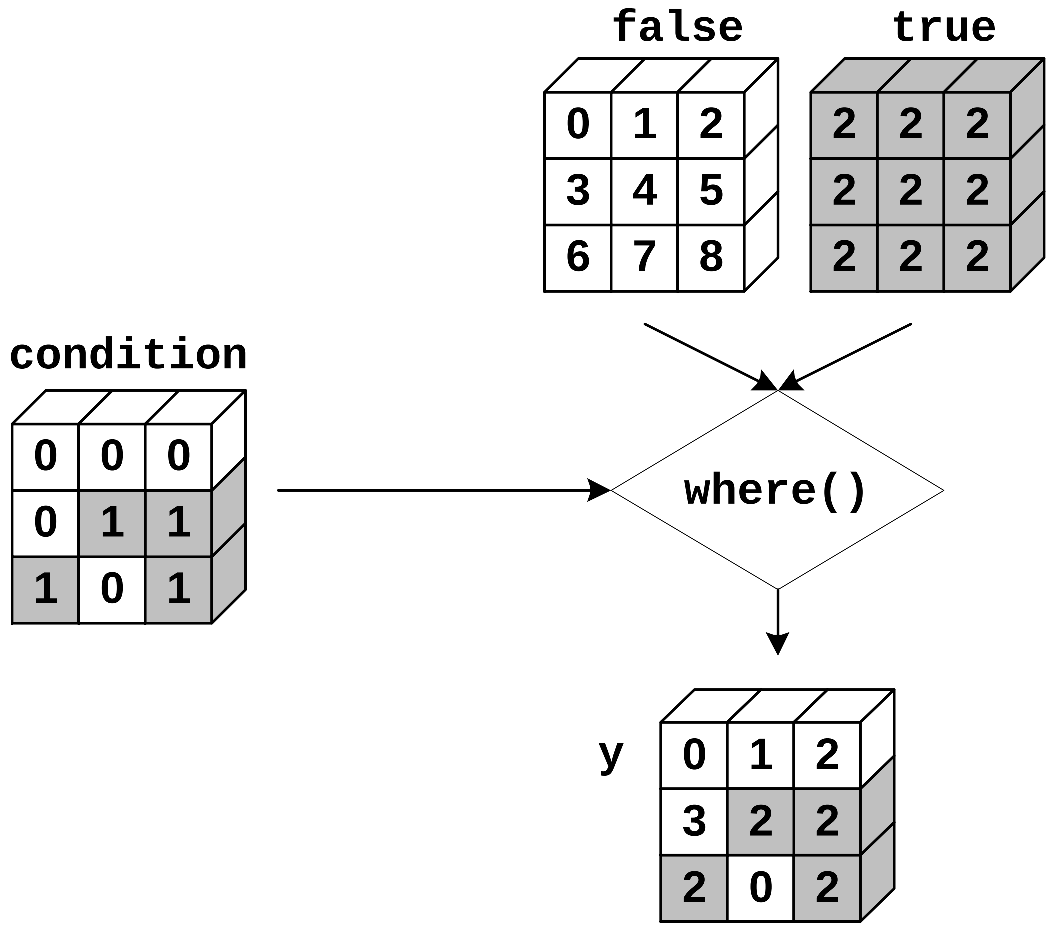
```
>>> a
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])

>>> lt10 = less(a,10)
>>> lt10
array([[1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> choose(lt10,(a,10))
array([[10, 10, 10, 10, 10],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
```

CLIP LOWER AND UPPER VALUES

```
>>> lt = less(a,10)
>>> gt = greater(a,15)
>>> choice = lt + 2 * gt
>>> choice
array([[1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0],
       [2, 2, 2, 2, 2]])
>>> choose(choice,(a,10,15))
array([[10, 10, 10, 10, 10],
       [10, 11, 12, 13, 14],
       [15, 15, 15, 15, 15]])
```

Array Functions – where()


```
>>> y = where(condition,false,true)
```

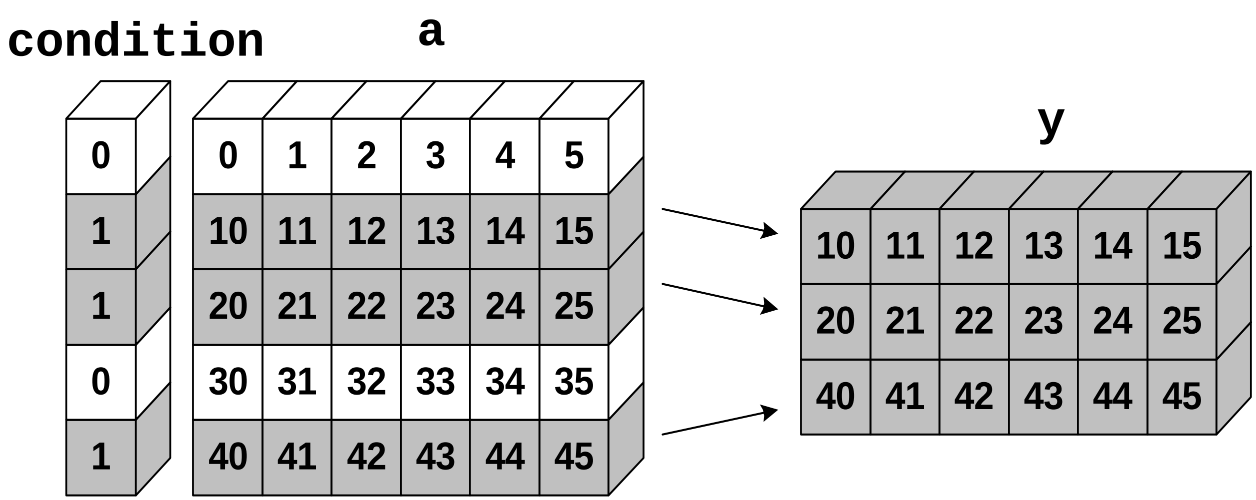


Array Functions – compress()

`compress(condition,a,axis=-1)`

Create an array from the slices (or elements) of `a` that correspond to the elements of `condition` that are "true". `condition` must not be longer than the indicated `axis` of `a`.

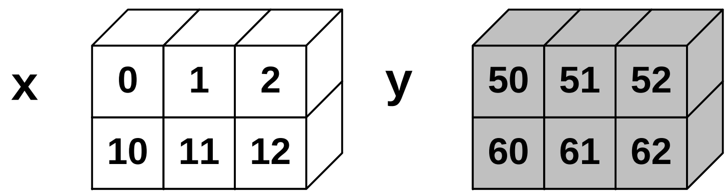
```
>>> compress(condition,a,0)
```



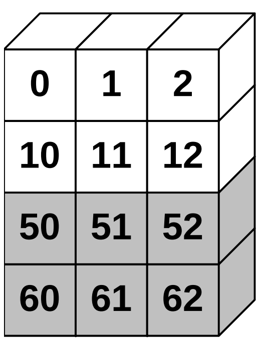
Array Functions – `concatenate()`

`concatenate((a0,a1,…,aN),axis=0)`

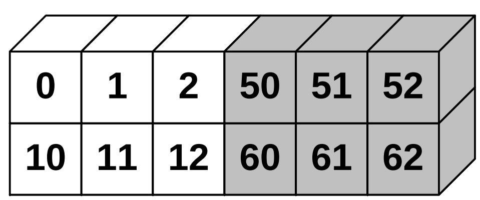
The input arrays `(a0,a1,…,aN)` will be concatenated along the given `axis`. They must have the same shape along every axis *except* the one given.



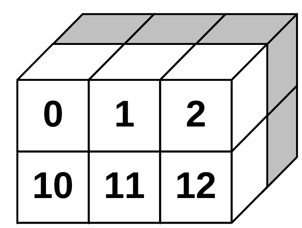

```
>>> concatenate((x,y))
```



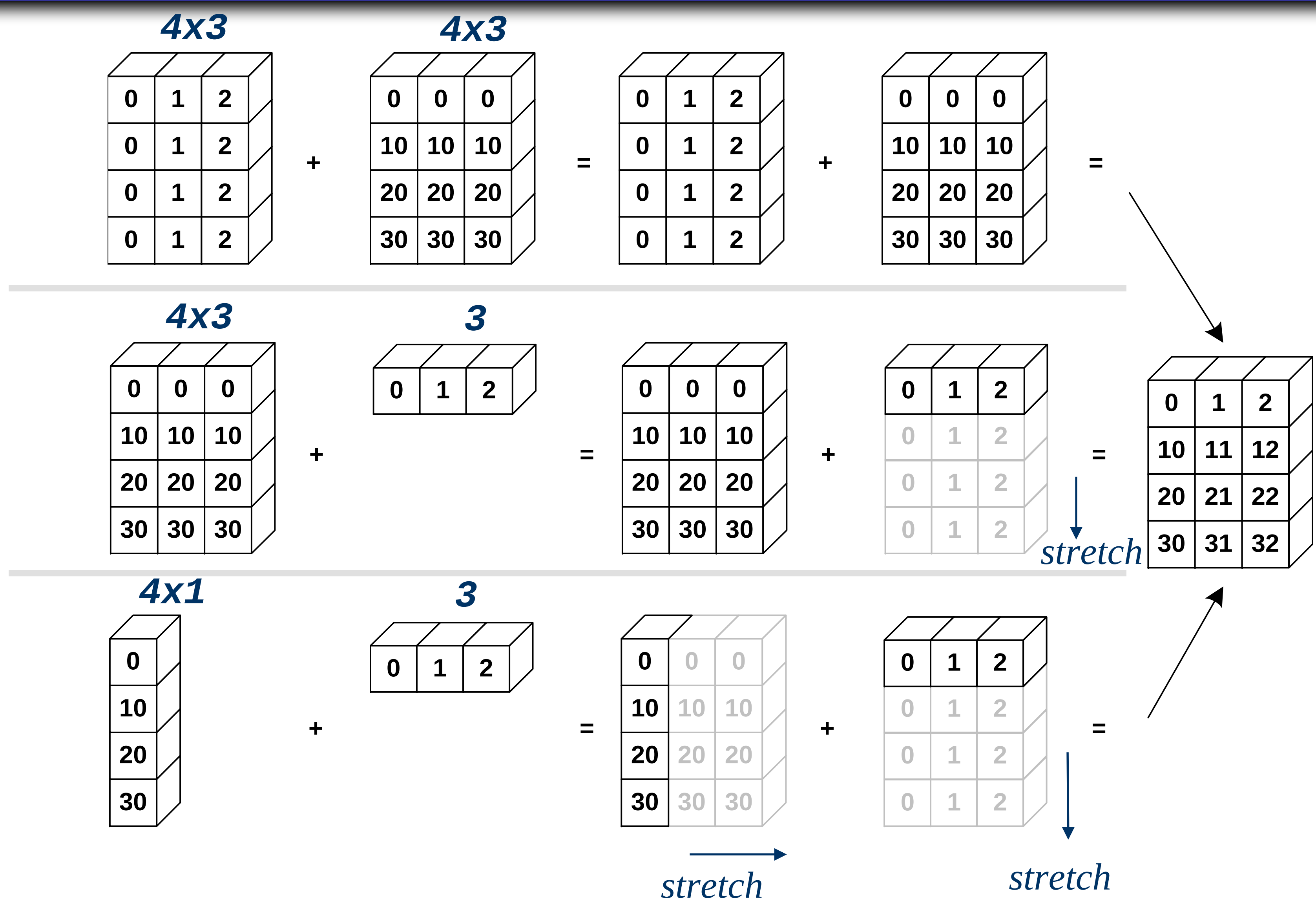

```
>>> concatenate((x,y),1)
```




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

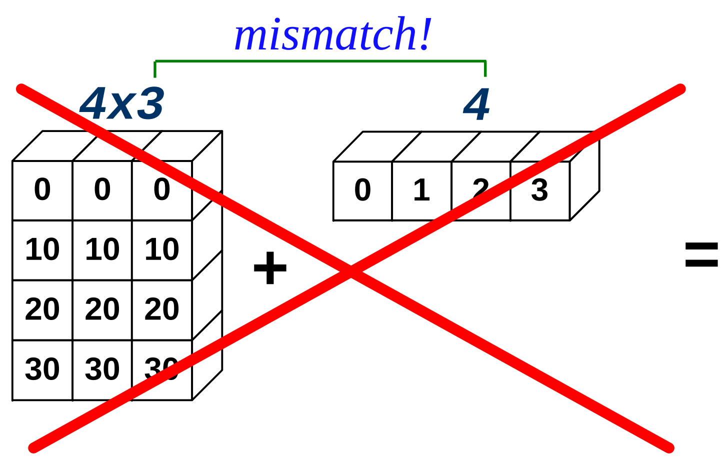


Array Broadcasting



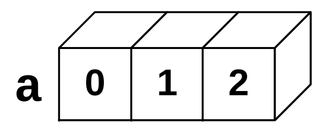
Broadcasting Rules

The *trailing* axes of both arrays must either be 1 or have the same size for broadcasting to occur. Otherwise, a "`ValueError: frames are not aligned`" exception is thrown.



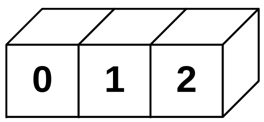
NewAxis

`NewAxis` is a special index that inserts a new axis in the array at the specified location. Each `NewAxis` increases the arrays dimensionality by 1.



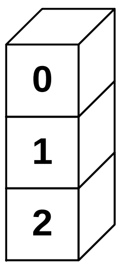
1 X 3

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



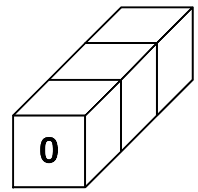
3 X 1

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



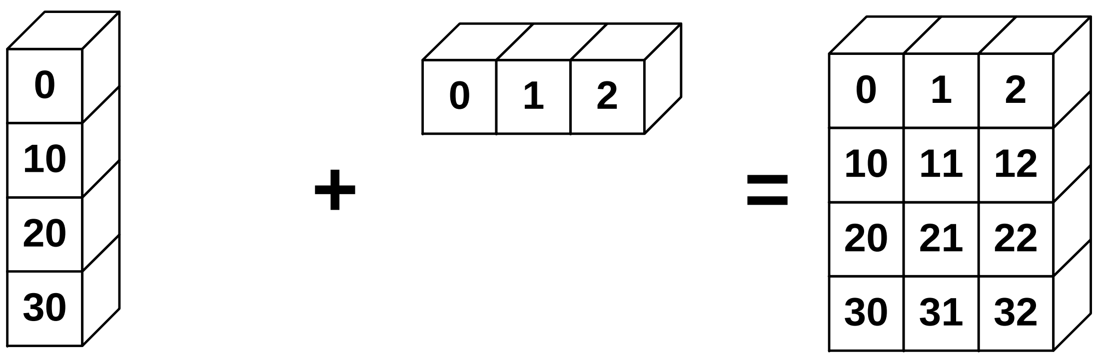
3 X 1 X 1

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



NewAxis in Action

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



enthought ®

Pickling

When pickling arrays, **use binary storage** when possible to save space.

```
>>> a = zeros((100,100),Float32)
# total storage
>>> a.itemsize()*len(a.flat)
40000
# standard pickling balloons 4x
>>> ascii = cPickle.dumps(a)
>>> len(ascii)
160061
# binary pickling is very nearly 1x
>>> binary = cPickle.dumps(a,1)
>>> len(binary)
40051
```

`Numeric` creates an intermediate string pickle when pickling arrays to a file resulting in a temporary 2x memory expansion. This can be very costly for huge arrays.

SciPy

Overview

- **Developed by Enthought and Partners**
  (Many thanks to Travis Oliphant and Pearu Peterson)
- **Open Source Python Style License**
- **Available at www.scipy.org**

CURRENT PACKAGES

- **Special Functions (scipy.special)**
- **Signal Processing (scipy.signal)**
- **Fourier Transforms (scipy.fftpack)**
- **Optimization (scipy.optimize)**
- **General plotting (scipy.[plt, xplt, gplt])**
- **Numerical Integration (scipy.integrate)**
- **Input/Output (scipy.io)**
- **Genetic Algorithms (scipy.ga)**
- **Statistics (scipy.stats)**
- **Distributed Computing (scipy.cow)**
- **Fast Execution (weave)**
- **Clustering Algorithms (scipy.cluster)**

Basic Environment

enthought ®

BYU

CONVENIENCE FUNCTIONS

```
>>> info(linspace)

 linspace(start, stop, num=50, endpoint=1, retstep=0)

Evenly spaced samples.

Return num evenly spaced samples from start to stop.  If endpoint=1 then
last sample is stop. If retstep is 1 then return the step value used.
```

`info` help system for scipy

```
>>> linspace(-1,1,5)
array([-1. , -0.5,  0. ,  0.5,  1. ])
>>> r_[-1:1:5j]
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

`linspace` get equally spaced points.

`r_[]` also does this (shorthand)

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

`logspace` get equally spaced points in log10 domain

```
>>> info(logspace)

 logspace(start, stop, num=50, endpoint=1)

Evenly spaced samples on a logarithmic scale.

Return num evenly spaced samples from 10**start to 10**stop.  If
endpoint=1 then last sample is 10**stop.
```

Basic Environment

CONVENIENCE FUNCTIONS

`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[0:5,0:5]
>>> x
array([[0],
       [1],
       [2],
       [3],
       [4]])
>>> y
array([       [0, 1, 2, 3, 4]])
>>> print x+y
[[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]
 [4 5 6 7 8]]
```

Basic Environment

enthought ®

BYU

CONVENIENT MATRIX GENERATION AND MANIPULATION

```
>>> A = mat('1,2,4;4,5,6;7,8,9')

>>> print A
Matrix([[1, 2, 4],
       [2, 5, 3],
       [7, 8, 9]])

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

>>> print A*A.I
Matrix([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

>>> print A.T
Matrix([[1, 2, 7],
       [2, 5, 8],
       [4, 3, 9]])
```

Simple creation of matrix with ";" meaning row separation

Matrix Power

Matrix Multiplication and Matrix Inverse

Matrix Transpose

enthought ®

More Basic Functions

TYPE HANDLING

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

SHAPE MANIPULATION

squeeze	vstack	split
atleast_1d	hstack	hsplit
atleast_2d	column_stack	vsplit
atleast_3d	dstack	dsplit
apply_over_ axes	expand_dims	apply_along_ axis

OTHER USEFUL FUNCTIONS

select	unwrap	roots
extract	sort_complex	poly
insert	trim_zeros	any
fix	fliplr	all
mod	flipud	disp
amax	rot90	unique
amin	eye	extract
ptp	diag	insert
sum	factorial	nansum
cumsum	factorial2	nanmax
prod	comb	nanargmax
cumprod	pade	nanargmin
diff	derivative	nanmin
angle	limits.XXXX	

Input and Output

scipy.io --- Raw data transfer from other programs

Before you use capabilities of scipy.io be sure that Pickle or netcdf (from Konrad Hinsen's ScientificPython) might not serve you better!

- Flexible facility for reading numeric data from text files and writing arrays to text files
- File class that streamlines transfer of raw binary data into and out of Numeric arrays
- Simple facility for storing Python dictionary into a module that can be imported with the data accessed as attributes of the module
- Compatibility functions for reading and writing MATLB .mat files
- Utility functions for packing bit arrays and byte swapping arrays

Input and Output

scipy.io --- Reading and writing ASCII files

textfile.txt

Student	Test1	Test2	Test3	Test4
Jane	98.3	94.2	95.3	91.3
Jon	47.2	49.1	54.2	34.7
Jim	84.2	85.3	94.1	76.4

Read from column 1 to the end

Read from line 3 to the end

```
>>> a = io.read_array('textfile.txt',columns=(1,-1),lines=(3,-1))

>>> print a
[[ 98.3  94.2  95.3  91.3]
 [ 47.2  49.1  54.2  34.7]
 [ 84.2  85.3  94.1  76.4]]
>>> b = io.read_array('textfile.txt',columns=(1,-2),lines=(3,-2))
>>> print b
[[ 98.3  95.3]
 [ 84.2  94.1]]
```

Read from column 1 to the end every second column

Read from line 3 to the end every second line

Input and Output

scipy.io --- Reading and writing raw binary files

fid = fopen(file_name, permission='rb', format='n')

Class for reading and writing binary files into Numeric arrays.

•file_name	The complete path name to the file to open.
•permission	Open the file with given permissions: ('r', 'w', 'a') for reading, writing, or appending. This is the same as the mode argument in the builtin open command.
•format	The byte-ordering of the file: (['native', 'n'], ['ieee-le', 'l'], ['ieee-be', 'b']) for native, little-endian, or big-endian.

Methods

read	read data from file and return Numeric array
write	write to file from Numeric array
fort_read	read Fortran-formatted binary data from the file.
fort_write	write Fortran-formatted binary data to the file.
rewind	rewind to beginning of file
size	get size of file
seek	seek to some position in the file
tell	return current position in file
close	close the file

Input and Output

scipy.io --- Making a module out of your data

Problem: You'd like to quickly save your data and pick up again where you left on another machine or at a different time.

Solution: Use io.save(<filename>,<dictionary>)
To load the data again use import <filename>

SAVING ALL VARIABLES

```
>>> io.save('allvars',globals())
                later
>>> from allvars import *
```

SAVING A FEW VARIABLES

```
>>> io.save('fewvars',{'a':a,'b':b)
                later
>>> import fewvars
>>> olda = fewvars.a
>>> oldb = fewvars.b
```

enthought ®

Polynomials

poly1d --- One dimensional polynomial class

- p = poly1d(<coefficient array>)
- p.roots (p.r) are the roots
- p.coefficients (p.c) are the coefficients
- p.order is the order
- p[n] is the coefficient of x^n
- p(val) evaulates the polynomial at val
- p.integ() integrates the polynomial
- p.deriv() differentiates the polynomial
- Basic numeric operations (+,-,/,*) work
- Acts like p.c when used as an array
- Fancy printing

```
>>> p = poly1d([1,-2,4])
>>> print p
 2
x - 2 x + 4

>>> g = p**3 + p*(3-2*p)
>>> print g
 6     5      4      3      2
x - 6 x + 25 x - 51 x + 81 x - 58 x +
44

>>> print g.deriv(m=2)
    4       3       2
30 x - 120 x + 300 x - 306 x + 162

>>> print p.integ(m=2,k=[2,1])
         4          3     2
0.08333 x - 0.3333 x + 2 x + 2 x + 1

>>> print p.roots
[ 1.+1.7321j  1.-1.7321j]

>>> print p.coeffs
[ 1 -2  4]
```

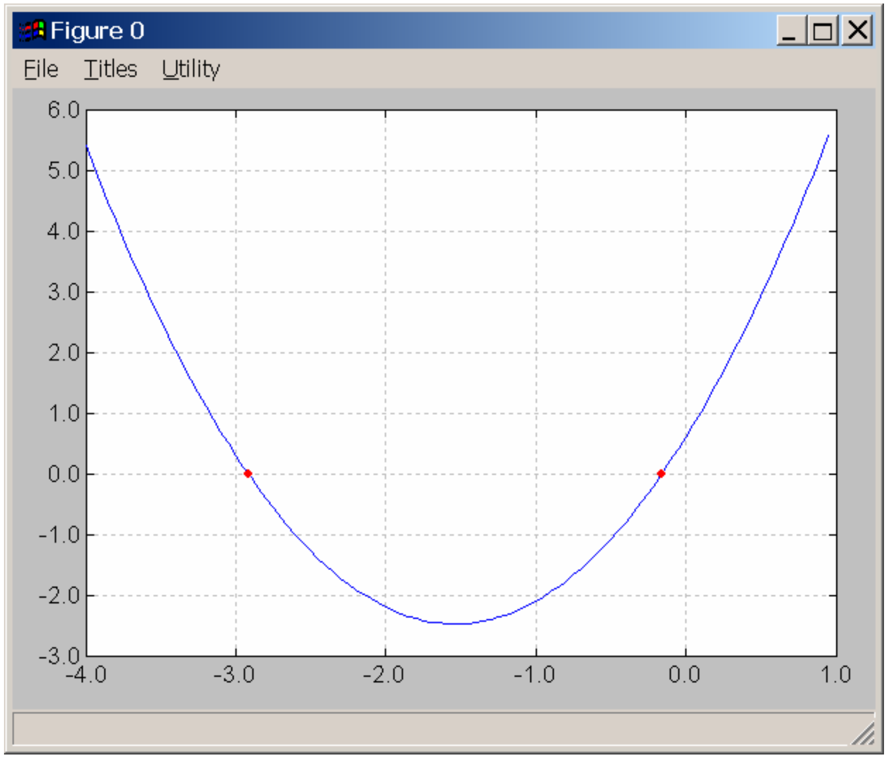
Polynomials

enthought ®

BYU

FINDING THE ROOTS OF A POLYNOMIAL

```
>>> p = poly1d([1.3,4,.6])
>>> print p
     2
1.3 x + 4 x + 0.6
>>> x = r_[-4:1:0.05]
>>> y = p(x)
>>> plt.plot(x,y,'-')
>>> plt.hold('on')
>>> r = p.roots
>>> s = p(r)
>>> r
array([-0.15812627, -2.9187968 ])
>>> plt.plot(r.real,s.real,'ro')
```

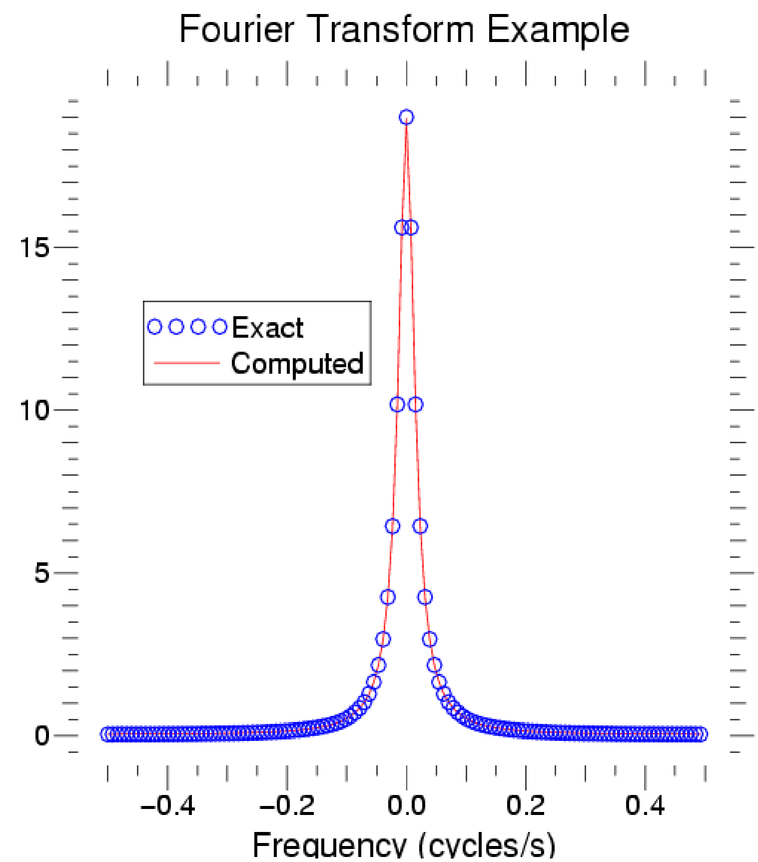


FFT

enthought ®

scipy.fft --- FFT and related functions

```
>>> n = fftfreq(128)*128
>>> f = fftfreq(128)
>>> ome = 2*pi*f
>>> x = (0.9)**abs(n)
>>> X = fft(x)
>>> z = exp(1j*ome)
>>> Xexact = (0.9**2 - 1)/0.9*z / (z-
0.9) / (z-1/0.9)
>>> xplt.plot(fftshift(f),
fftshift(X.real),'r',fftshift(f),
fftshift(Xexact.real),'bo')
>>> xplt.expand_limits(10)
>>> xplt.title('Fourier Transform
Example')
>>> xplt.xlabel('Frequency
(cycles/s)')
>>> xplt.legend(['Computed','Exact'])
Click on point for lower left
coordinate
>>> xplt.eps('figures/fft_example1')
```



Linear Algebra

scipy.linalg --- FAST LINEAR ALGEBRA

- **Uses ATLAS if available --- very fast**
- **Low-level access to BLAS and LAPACK routines in modules `linalg.fblas`, and `linalg.flapack` (FORTRAN order)**
- **High level matrix routines**
  - **Linear Algebra Basics:** `inv`, `solve`, `det`, `norm`, `lstsq`, `pinv`
  - **Decompositions:** `eig`, `lu`, `svd`, `orth`, `cholesky`, `qr`, `schur`
  - **Matrix Functions:** `expm`, `logm`, `sqrtm`, `cosm`, `coshm`, `funm` **(general matrix functions)**

enthought ®

BYU

Special Functions

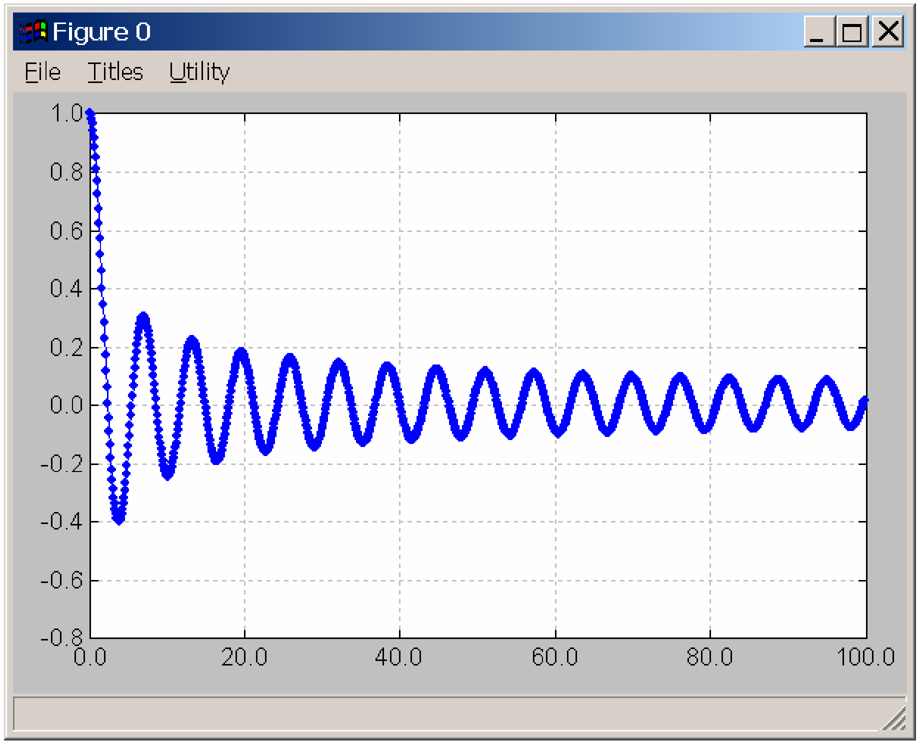
scipy.special

Includes over 200 functions:

Airy, Elliptic, Bessel, Gamma, HyperGeometric, Struve, Error, Orthogonal Polynomials, Parabolic Cylinder, Mathieu, Spheroidal Wave, Kelvin

FIRST ORDER BESSEL EXAMPLE

```
#environment setup
>>> import gui_thread
>>> gui_thread.start()
>>> from scipy import *
>>> import scipy.plt as plt


>>> x = r_[0:100:0.1]
>>> j0x = special.j0(x)
>>> plt.plot(x,j0x)
```



Special Functions

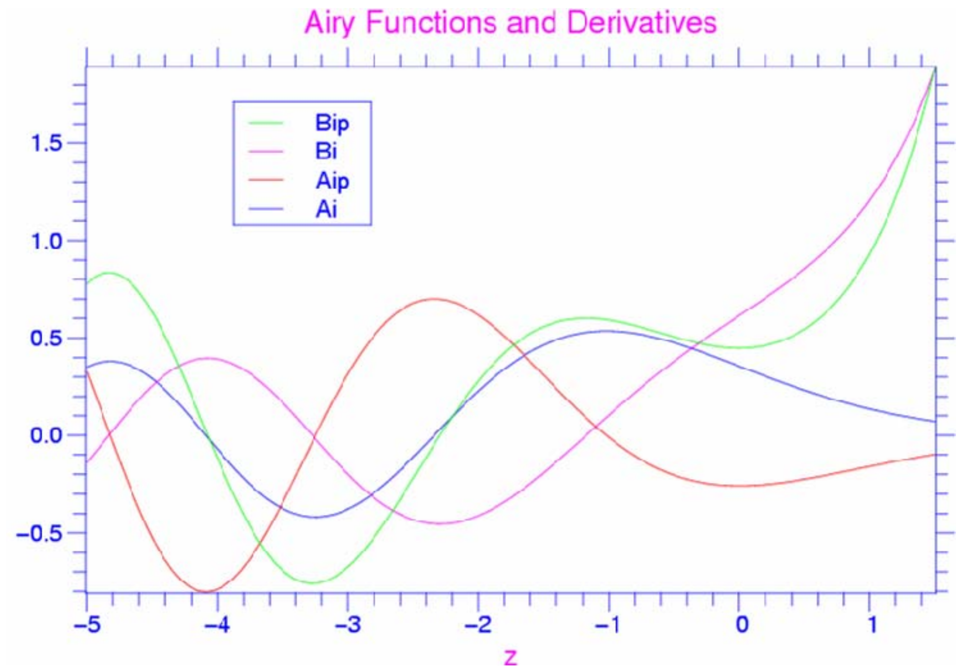
enthought ®

BYU

scipy.special

AIRY FUNCTIONS EXAMPLE

```
>>> z = r_[-5:1.5:100j]
>>> vals = special.airy(z)
>>> xplt.figure(0, frame=1,
        color='blue')
>>> xplt.matplot(z,vals)
>>> xplt.legend(['Ai', 'Aip',
        ‘Bi‘,'Bip'],
   color='blue')
>>> xplt.xlabel('z',
        color='magenta')
>>> xplt.title('Airy
   Functions and
   Derivatives‘)
```



Special Functions

scipy.special --- Vectorizing a function

- All of the special functions can operate over an array of data (they are “vectorized”) and follow the broadcasting rules.
- At times it is easy to write a scalar version of a function but hard to write the “vectorized” version.
- scipy.vectorize() will take any Python callable object (function, method, etc., and return a callable object that behaves like a “vectorized” version of the function)
- Similar to list comprehensions in Python but more general (N-D loops and broadcasting for multiple inputs).

Special Functions

enthought ®

scipy.special --- Vectorizing a function

Example

```
# 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
>>> sinc([1.3,1.5])
TypeError: can't multiply sequence
to non-int
```

Solution

```
>>> vsinc = vectorize(sinc)
>>> vsinc([1.3,1.5])
array([-0.1981, -0.2122])
```

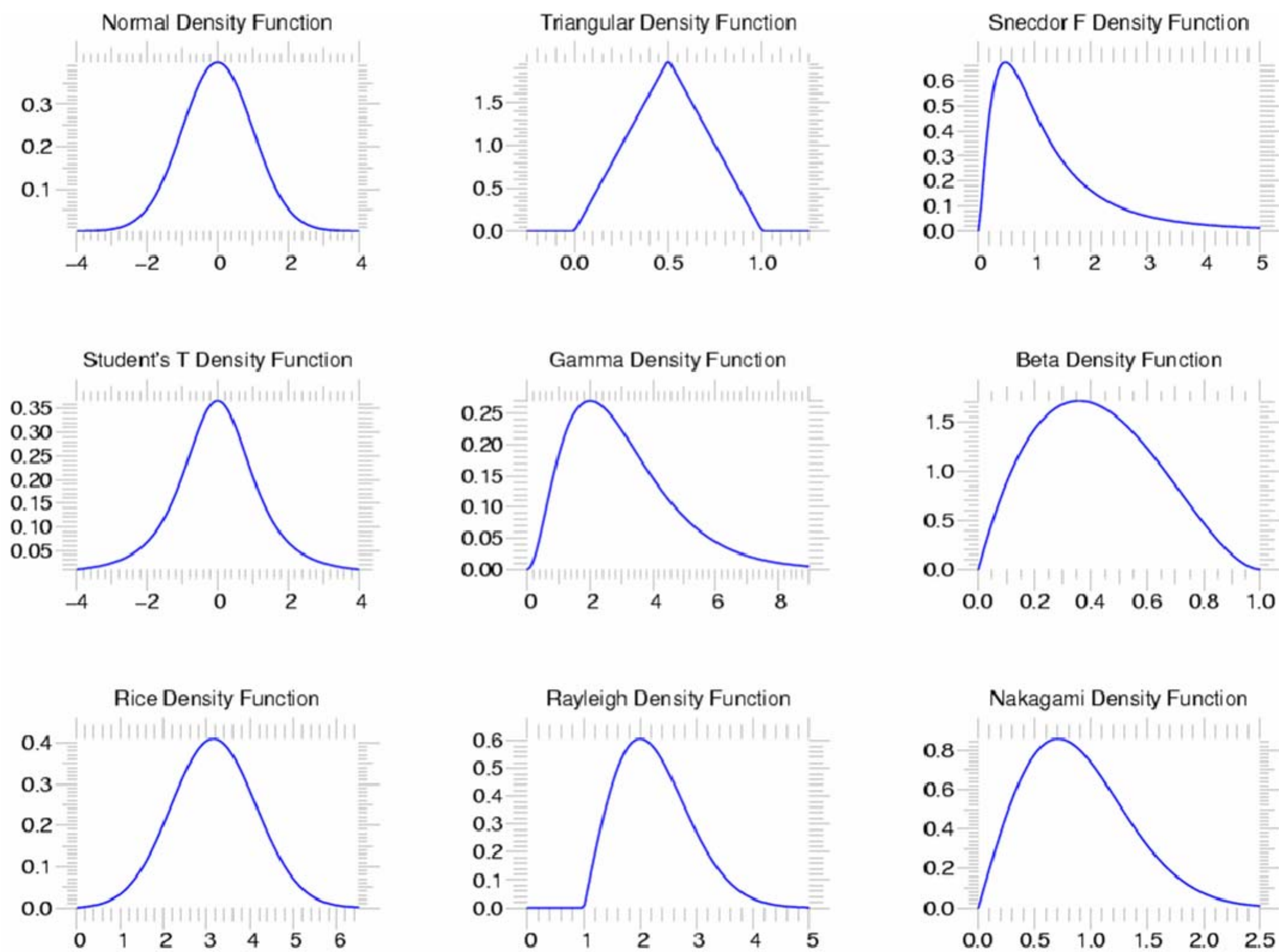
Statistics

enthought ®

BYU

scipy.stats --- Continuous Distributions

over 80 continuous distributions!



Methods

pdf

cdf

rvs

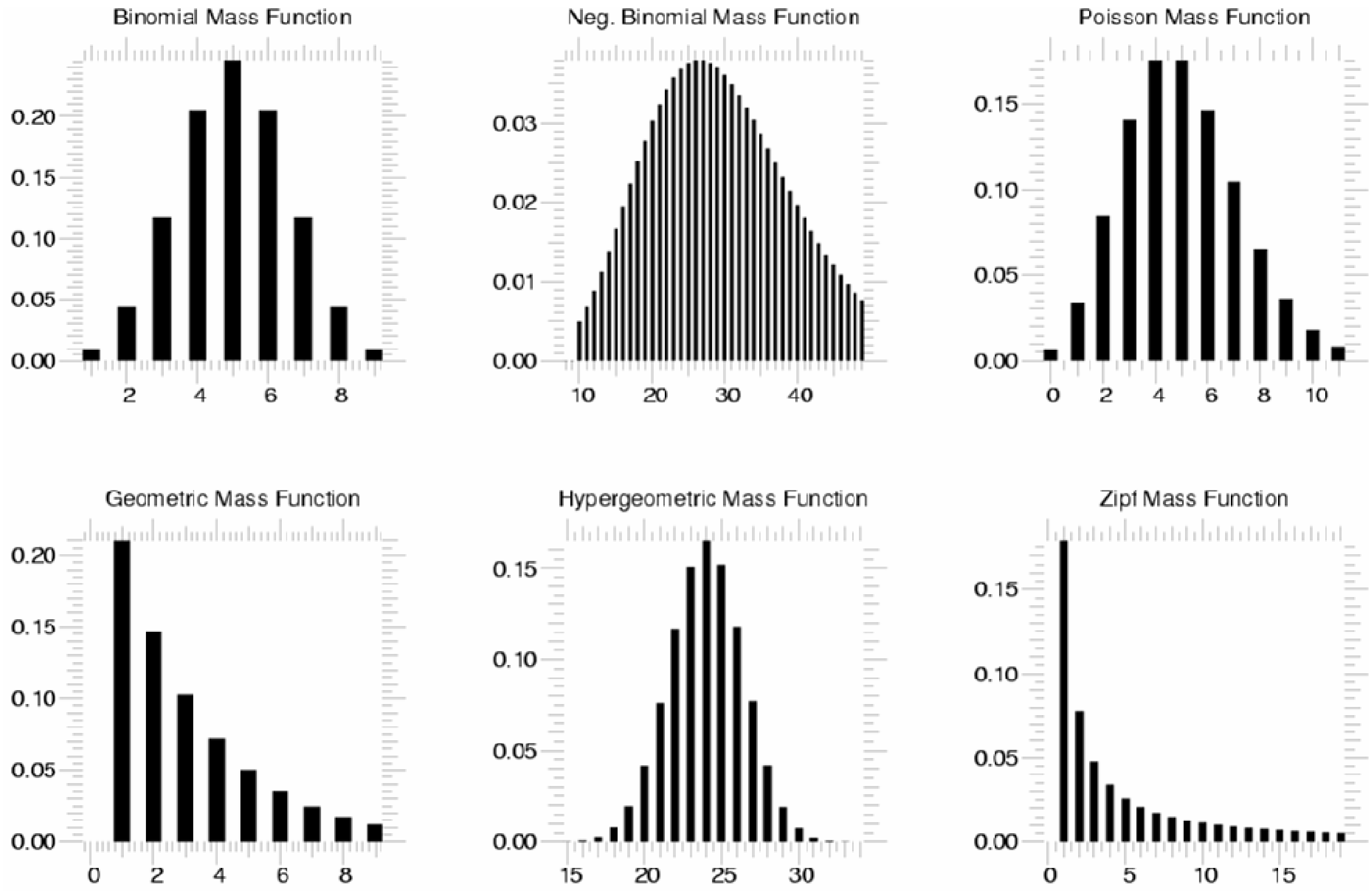
ppf

stats

Statistics

scipy.stats --- Discrete Distributions

10 standard discrete distributions (plus any arbitrary finite RV)



Methods

pdf

cdf

rvs

ppf

stats

Statistics

scipy.stats --- Basic Statistical Calculations for samples

• `stats.mean` (also `mean`)	compute the sample mean
• `stats.std` (also `std`)	compute the sample standard deviation
• `stats.var`	sample variance
• `stats.moment`	sample central moment
• `stats.skew`	sample skew
• `stats.kurtosis`	sample kurtosis

Interpolation

scipy.interpolate --- General purpose Interpolation

- **1-d linear Interpolating Class**
  - Constructs callable function from data points
  - Function takes vector of inputs and returns linear interpolants
- **1-d and 2-d spline interpolation (FITPACK)**
  - Splines up to order 5
  - Parametric splines

Integration

scipy.integrate --- General purpose Integration

- **Ordinary Differential Equations (ODE)**

  `integrate.odeint`, `integrate.ode`

- **Samples of a 1-d function**

  `integrate.trapz` (trapezoidal Method), `integrate.simps` (Simpson Method), `integrate.romb` (Romberg Method)

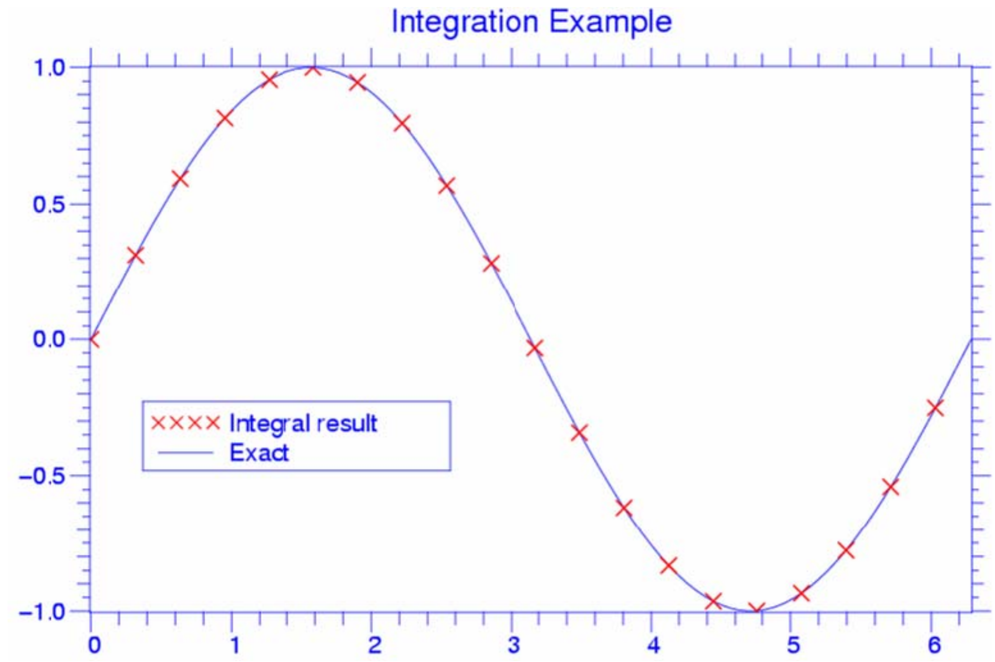
- **Arbitrary callable function**

  `integrate.quad` (general purpose), `integrate.dblquad` (double integration), `integrate.tplquad` (triple integration), `integrate.fixed_quad` (fixed order Gaussian integration), `integrate.quadrature` (Gaussian quadrature to tolerance), `integrate.romberg` (Romberg)

Integration

scipy.integrate --- Example

```
>>> def func(x):
    return integrate.quad(cos,0,x)[0]
>>> vecfunc = vectorize(func)

>>> x = r_[0:2*pi:100j]
>>> x2 = x[::5]
>>> y = sin(x)
>>> y2 = vecfunc(x2)
>>> xplt.plot(x,y,x2,y2,'rx')
```



Signal Processing

scipy.signal --- Signal and Image Processing

What's Available?

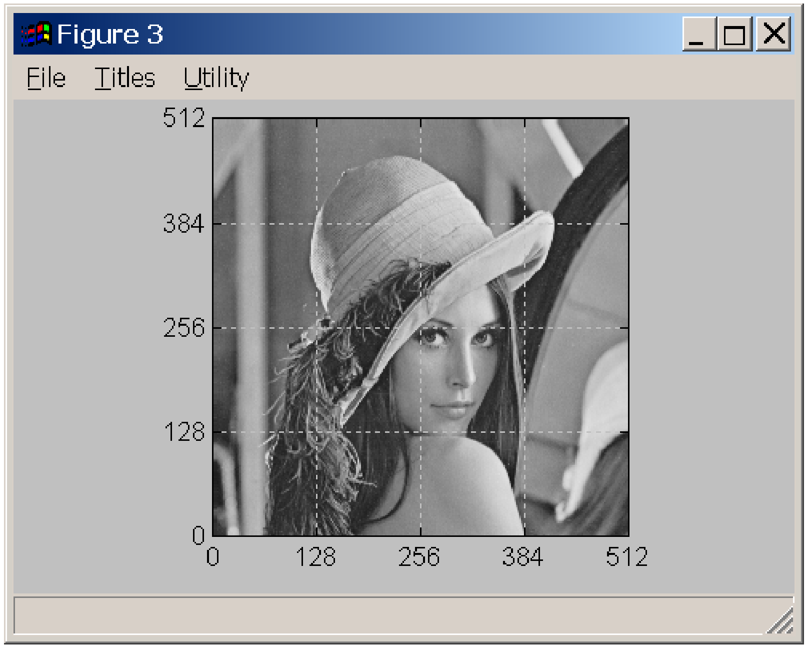
- **Filtering**
    - General 2-D Convolution (more boundary conditions)
    - N-D convolution
    - B-spline filtering
    - N-D Order filter, N-D median filter, faster 2d version,
    - IIR and FIR filtering and filter design
- LTI systems
    - System simulation
    - Impulse and step responses
    - Partial fraction expansion

Image Processing

enthought ®

```
# Blurring using a median filter
>>> lena = lena()
>>> lena = lena.astype(Float32)
>>> plt.image(lena)
>>> fl = signal.medfilt2d(lena,[15,15])
>>> plt.image(fl)
```

LENA IMAGE



MEDIAN FILTERED IMAGE

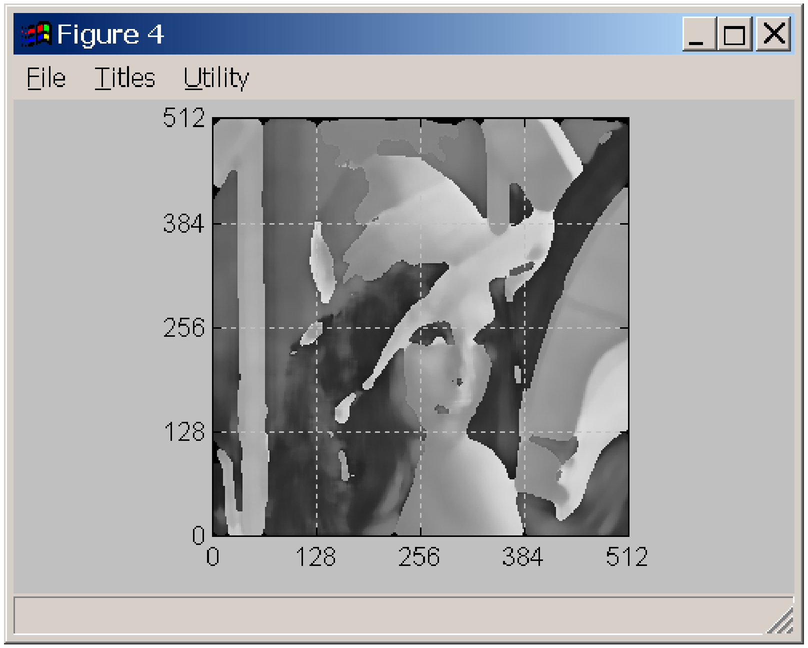


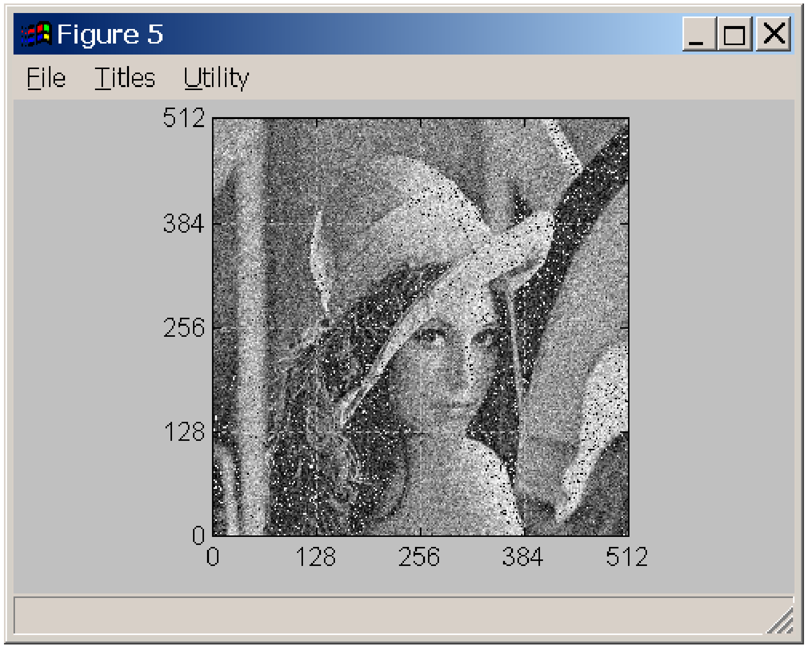
Image Processing

enthought ®

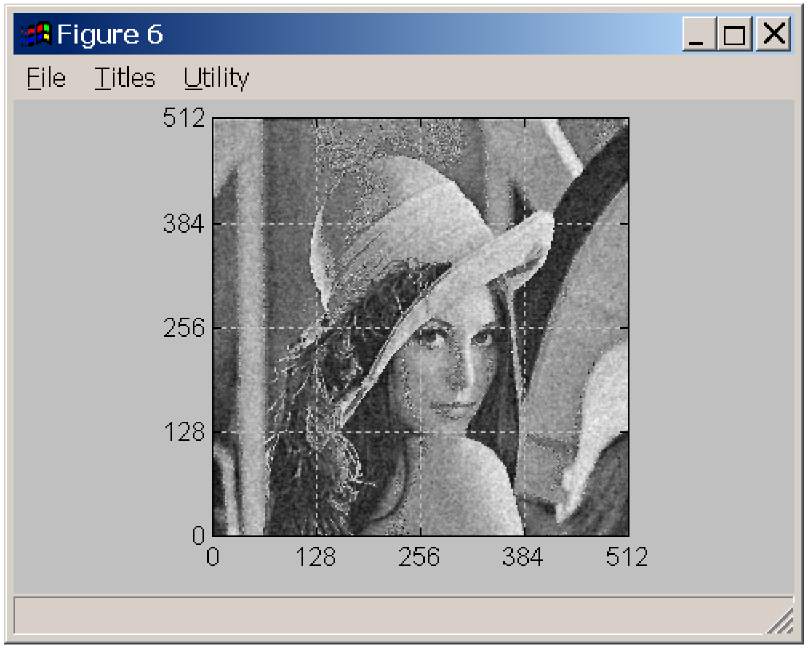
BYU

```
# Noise removal using wiener filter
>>> from scipy.stats import norm
>>> ln = lena + norm(0,32,shape(lena))
>>> cleaned = signal.wiener(ln)
>>> plt.plot(cleaned)
```

NOISY IMAGE



FILTERED IMAGE

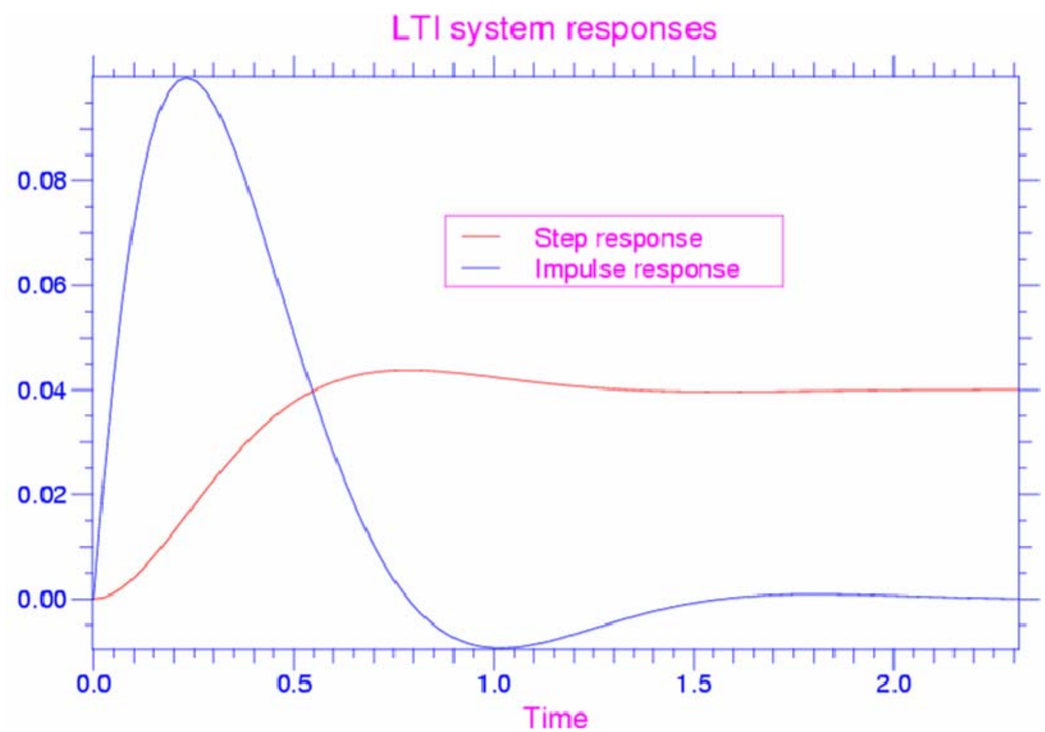


LTI Systems

enthought ®

$$H(s) = \frac{1}{s^2 + 6s + 25}$$

```
>>> b,a = [1],[1,6,25]
>>> ltisys = signal.lti(b,a)
>>> t,h = ltisys.impulse()
>>> t,s = ltisys.step()
>>> xplt.plot(t,h,t,s)
>>> xplt.legend(['Impulse
response','Step response'],
color='magenta')
```



Optimization

enthought ®

scipy.optimize --- unconstrained minimization and root finding

- **Unconstrained Optimization**

  `fmin` (Nelder-Mead simplex), **`fmin_powell`** (Powell's method), **`fmin_bfgs`** (BFGS quasi-Newton method), **`fmin_ncg`** (Newton conjugate gradient), **`leastsq`** (Levenberg-Marquardt), **`anneal`** (simulated annealing global minimizer), **`brute`** (brute force global minimizer), **brent** (excellent 1-D minimizer), **golden**, **bracket**

- **Constrained Optimization**

  `fmin_l_bfgs_b`, **`fmin_tnc`** (truncated newton code), **`fmin_cobyla`** (constrained optimization by linear approximation), **`fminbound`** (interval constrained 1-d minimizer)

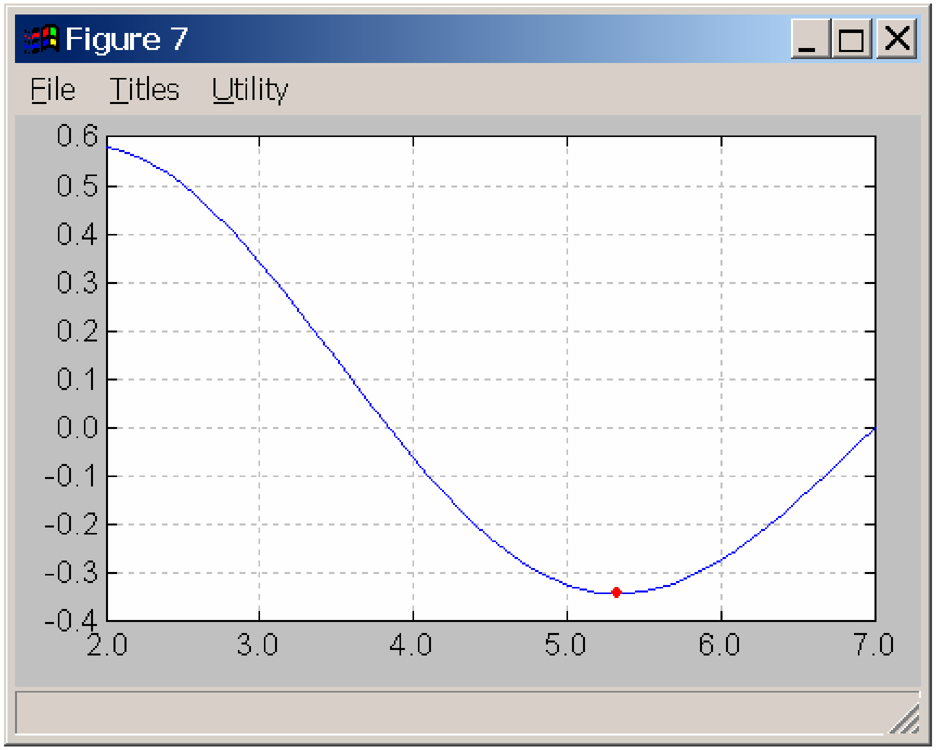
- **Root finding**

  `fsolve` (using MINPACK), **`brentq`**, **`brenth`**, **`ridder`**, **`newton`**, **`bisect`**, **`fixed_point`** (fixed point equation solver)

Optimization

EXAMPLE: MINIMIZE BESSEL FUNCTION

```
# minimize 1st order bessel
# function between 4 and 7
>>> from scipy.special import j1
>>> from scipy.optimize import \
    fminbound

>>> x = r_[2:7.1:.1]
>>> j1x = j1(x)
>>> plt.plot(x,j1x,'-')
>>> plt.hold('on')
>>> j1_min = fminbound(j1,4,7)
>>> plt.plot(x,j1_min,'ro')
```



Optimization

EXAMPLE: SOLVING NONLINEAR EQUATIONS

Solve the non-linear equations

$$3x_0 - \cos(x_1 x_2) + a = 0$$
$$x_0^2 - 81(x_1 + 0.1)^2 + \sin(x_2) + b = 0$$
$$e^{-x_0 x_1} + 20x_2 + c = 0$$

starting location for search

```
>>> def nonlin(x,a,b,c):
>>>     x0,x1,x2 = x
>>>     return [3*x0-cos(x1*x2)+ a,
>>>                 x0*x0-81*(x1+0.1)**2
                     + sin(x2)+b,
>>>                 exp(-x0*x1)+20*x2+c]
>>> a,b,c = -0.5,1.06,(10*pi-3.0)/3
>>> root = optimize.fsolve(nonlin,
               [0.1,0.1,-0.1],args=(a,b,c))
>>> print root
>>> print nonlin(root,a,b,c)
[ 0.5     0.     -0.5236]
[0.0, -2.231104190e-12, 7.46069872e-14]
```

Optimization

EXAMPLE: MINIMIZING ROSENBROCK FUNCTION

Rosenbrock function

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} 100\left(x_i - x_{i-1}^2\right)^2 + (1 - x_{i-1})^2 .$$

WITHOUT DERIVATIVE

```
>>> rosen = optimize.rosen
>>> import time
>>> x0 = [1.3,0.7,0.8,1.9,1.2]
>>> start = time.time()
>>> xopt = optimize.fmin(rosen,
x0, avegtol=1e-7)
>>> stop = time.time()
>>> print_stats(start, stop, xopt)
Optimization terminated successfully.
        Current function value: 0.000000
        Iterations: 316
        Function evaluations: 533
Found in 0.0805299282074 seconds
Solution: [ 1.  1.  1.  1.  1.]
Function value: 2.67775760157e-15
Avg. Error: 1.5323906899e-08
```

USING DERIVATIVE

```
>>> rosen_der = optimize.rosen_der
>>> x0 = [1.3,0.7,0.8,1.9,1.2]
>>> start = time.time()
>>> xopt = optimize.fmin_bfgs(rosen,
x0, fprime=rosen_der, avegtol=1e-7)
>>> stop = time.time()
>>> print_stats(start, stop, xopt)
Optimization terminated successfully.
        Current function value: 0.000000
        Iterations: 111
        Function evaluations: 266
        Gradient evaluations: 112
Found in 0.0521121025085 seconds
Solution: [ 1.  1.  1.  1.  1.]
Function value: 1.3739103475e-18
Avg. Error: 1.13246034772e-10
```

GA and Clustering

scipy.ga --- Basic Genetic Algorithm Optimization

Routines and classes to simplify setting up a genome and running a genetic algorithm evolution

scipy.cluster --- Basic Clustering Algorithms

- **Observation whitening** `cluster.vq.whiten`
- **Vector quantization** `cluster.vq.vq`
- **K-means algorithm** `cluster.vq.kmeans`

2D Plotting and Visualization

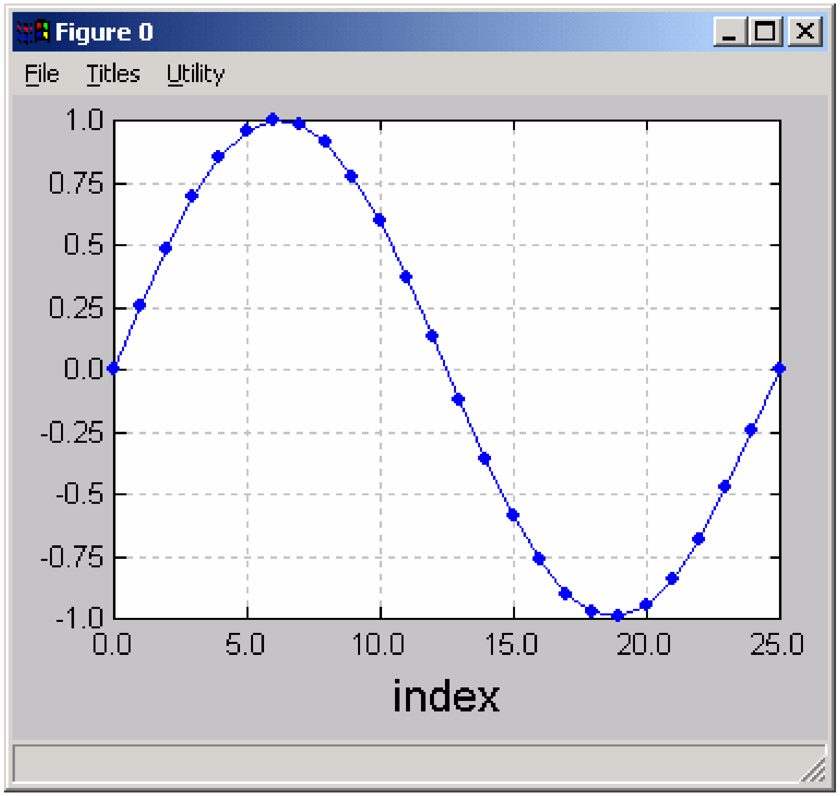
2D Plotting Overview

- Multiple interactive plots
- Plots and command line available simultaneously
- Easy one line plot commands for “everyday” analysis (Matlab-like)
- wxPython based
- Object oriented core

Scatter Plots

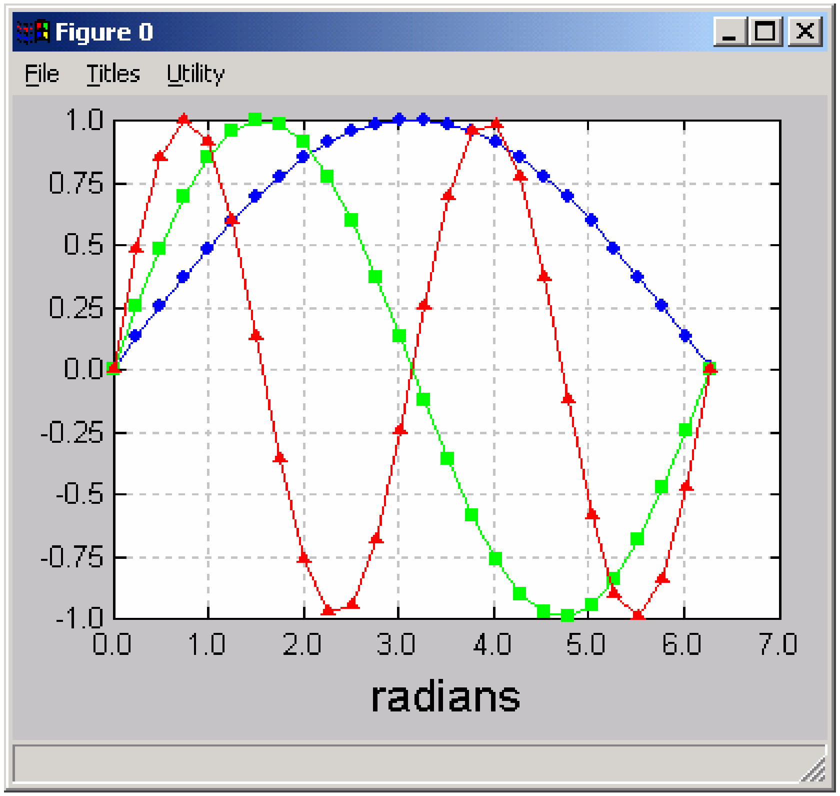
PLOT AGAINST INDICES

```
>>> plt.plot(y)
>>> plt.xtitle('index')
```



PLOT X VS. Y (multiple Y values)

```
>>> plot(x,y_group)
>>> plt.xtitle('radians')
```

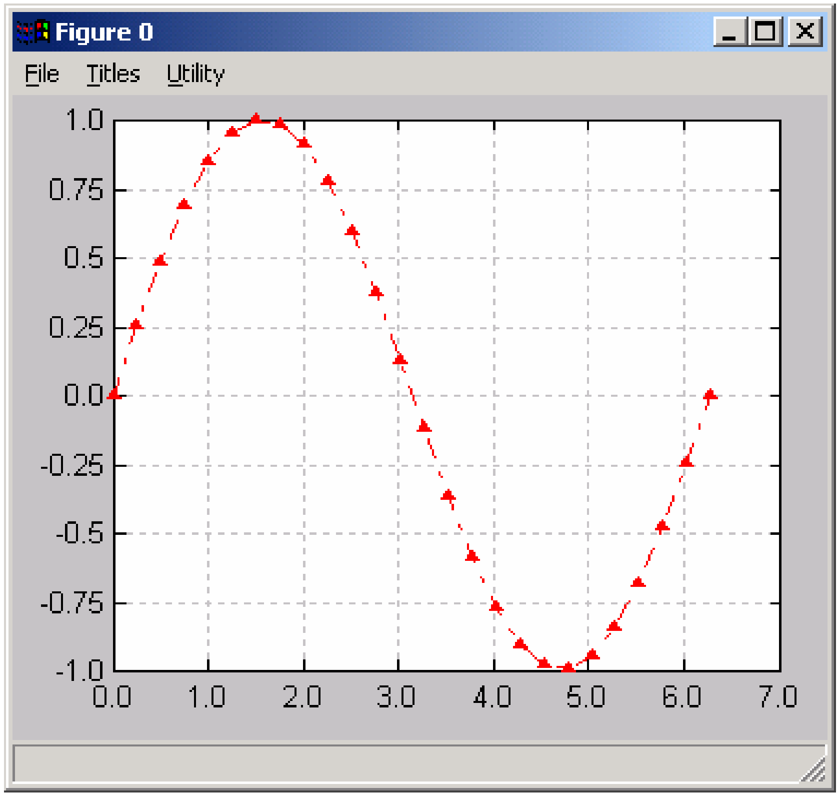


Scatter Plots

enthought ®

LINE FORMATTING

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



MULTIPLE PLOT GROUPS

```
>>> plot(x1,y1,'b-o',x2,y2)
>>> plt.yaxis([-2,2])
```

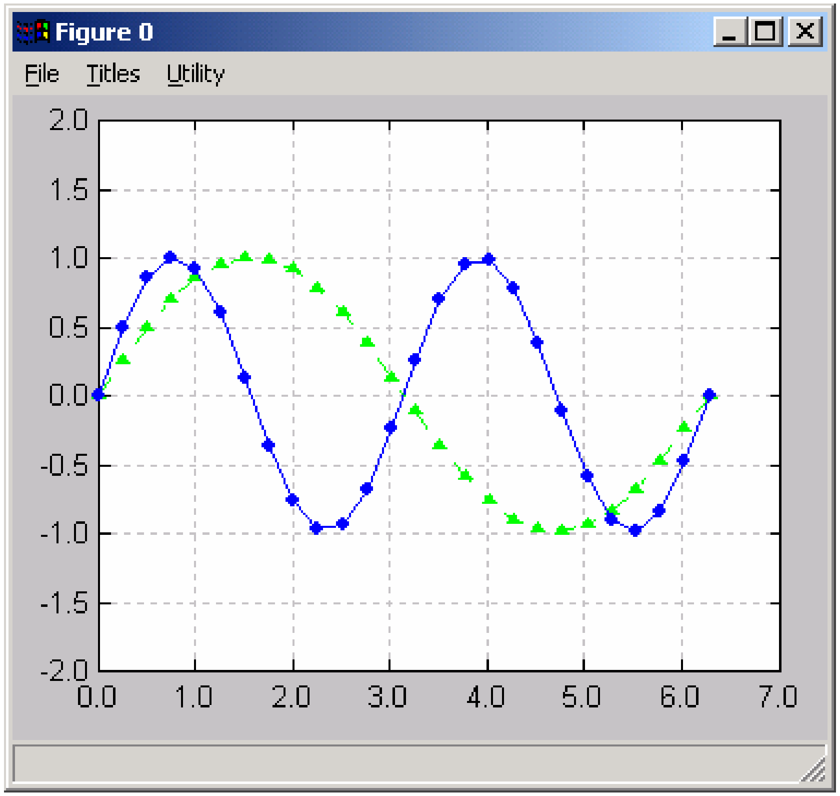
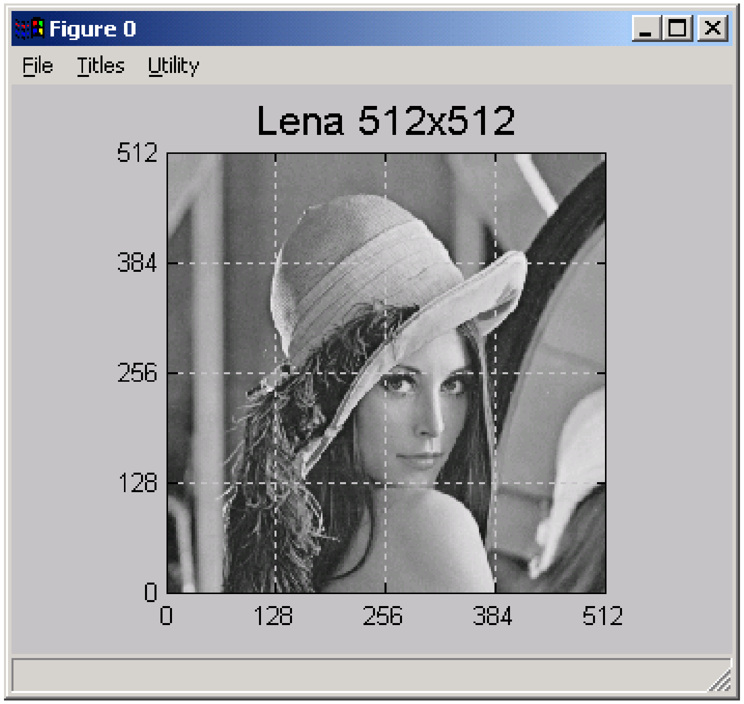


Image Display

enthought ®

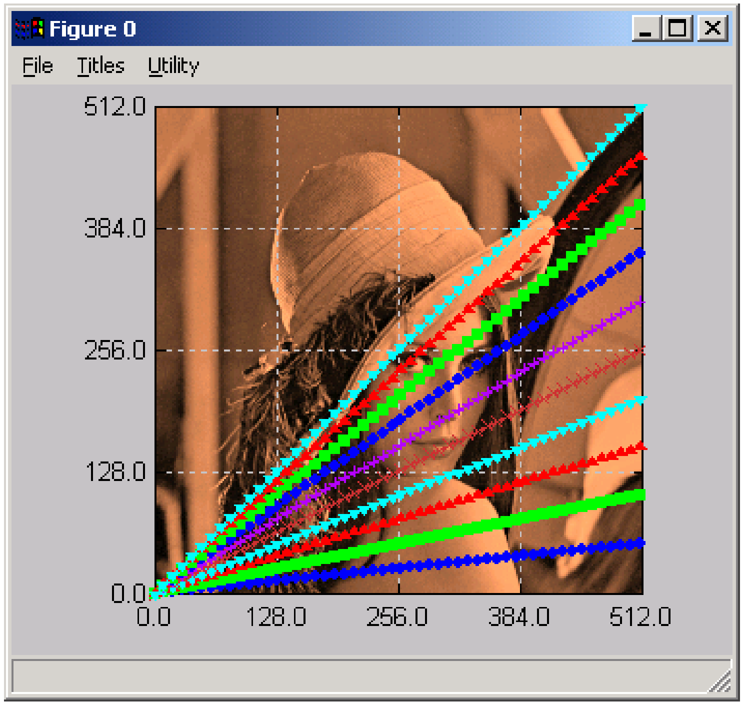
PLOT AGAINST INDICES

```
>>> plt.image(lena)
>>> plt.title('Lena 512x512')
```



PLOT X VS. Y (multiple Y values)

```
>>> plt.image(lena,
...             colormap='copper')
>>> plt.hold('on')
>>> plt.plot(x,lines)
```



Command Synopsis for `plt`

enthought®

BYU

PLOTTING

`plot(x,y,line_format,…)`

Create a scatter plot.

`image(img,x,y,colormap='grey')`

Display the `img` matrix.

WINDOW MANAGEMENT

`figure(which_one)`

Create a new window or activate and old one.

`current()`

Get handle to current window.

`close(which_one)`

Close current or specified window.

`save(file_name,format='png')`

Save plot to file.

TEXT

`title(text)` Place title above plot.

`xtitle(text)` Label x axis.

`ytitle(text)` Label y axis.

AXIS

`autoscale()`

Scale axes to data.

`grid(state=None)`

Toggle gridlines on and off.

`xaxis([lower,upper,interval])`

`yaxis([lower,upper,interval])`

Set the limits of an axis.

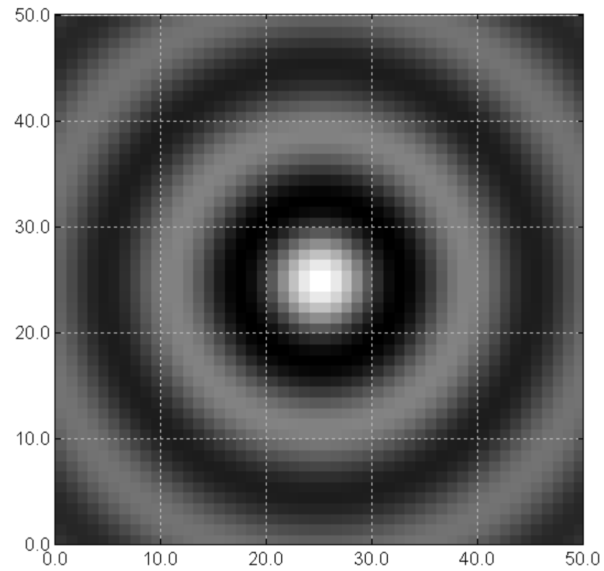
`axis(setting)`

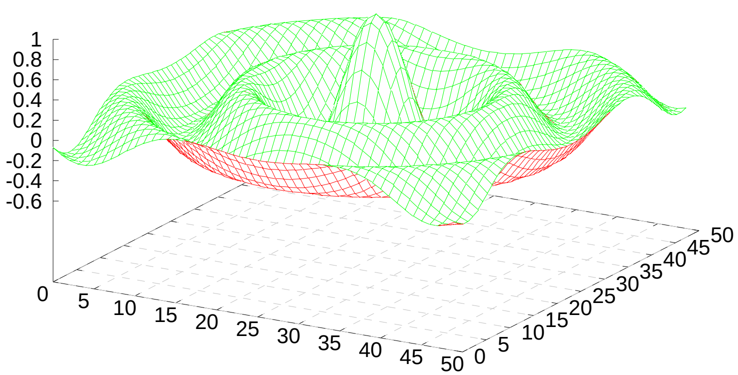
Specifies how axes values are calculated.

Surface plots with gplt

enthought ®

```
# Create 2d array where values
# are radial distance from
# the center of array.
>>> x = (arange(50.) - 24.5)/2.
>>> y = (arange(50.) - 24.5)/2.
>>> r = sqrt(x**2+y[:,NewAxis]**2)
# Calculate bessel function of
# each point in array.
>>> s=scipy.special.j0(r)

# Display image of Bessel function.
>>> plt.imagesc(s)

# Display surface plot.
>>> from scipy import gplt
>>> gplt.surf(s)
>>> gplt.hidden('remove')
```

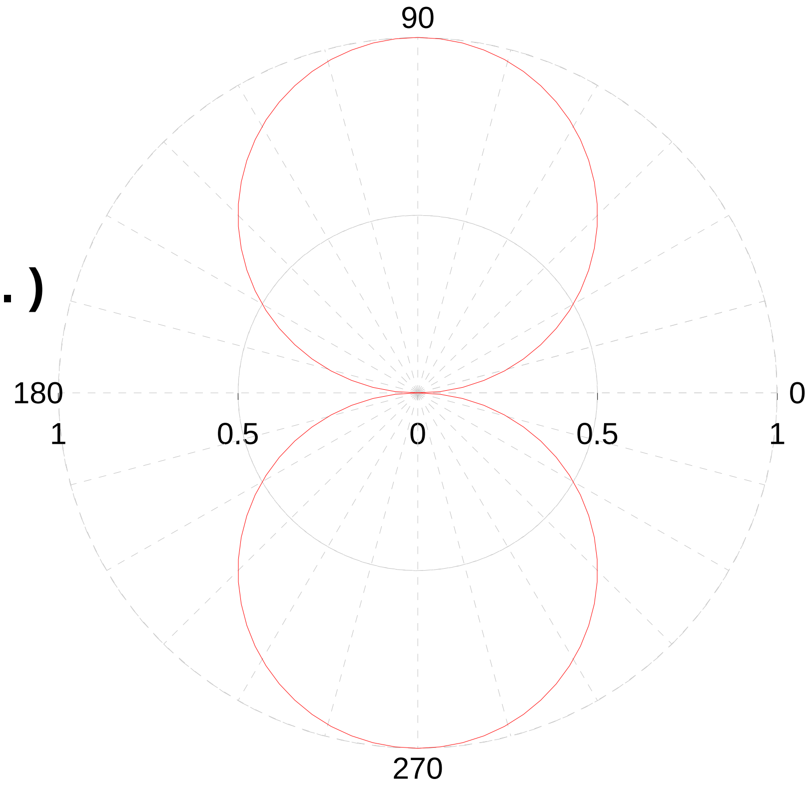




Polar Plots

enthought ®

```
# Create 2d array where values
# are radial distance from
# the center of array.
>>> angle = arange(101.)*(2*pi/100.)
>>> r = abs(sin(a))

# Generate the plot.
>>> gplt.polar(angle,r)
```



Other plotting libraries

enthought ®

- Chaco – new release in December.
- Matplotlib – Alternative plotting package that is fairly full featured and easy to use.