# Introduction to Scientific Computing with Python

Eric Jones
eric@enthought.com

Enthought, Inc.
www.enthought.com



## SciPy 2007 Conference


**http://www.scipy.org/SciPy2007**

**Aug 14th-18th in CalTech**

# 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**



# Who is using Python?

## SPACE TELESCOPE SCIENCE INSTITUTE

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

## HOLLYWOOD

Digital animation and special effects:
Industrial Light and Magic
Imageworks
Tippett Studios
Disney
Dreamworks

## PETROLEUM INDUSTRY

Geophysics and exploration tools:
ConocoPhillips, Shell

## GOOGLE

One of top three languages used at Google along with C++ and Java. Guido works there.

## PAINT SHOP PRO 9

Scripting Engine for JASC PaintShop Pro photo-editing software

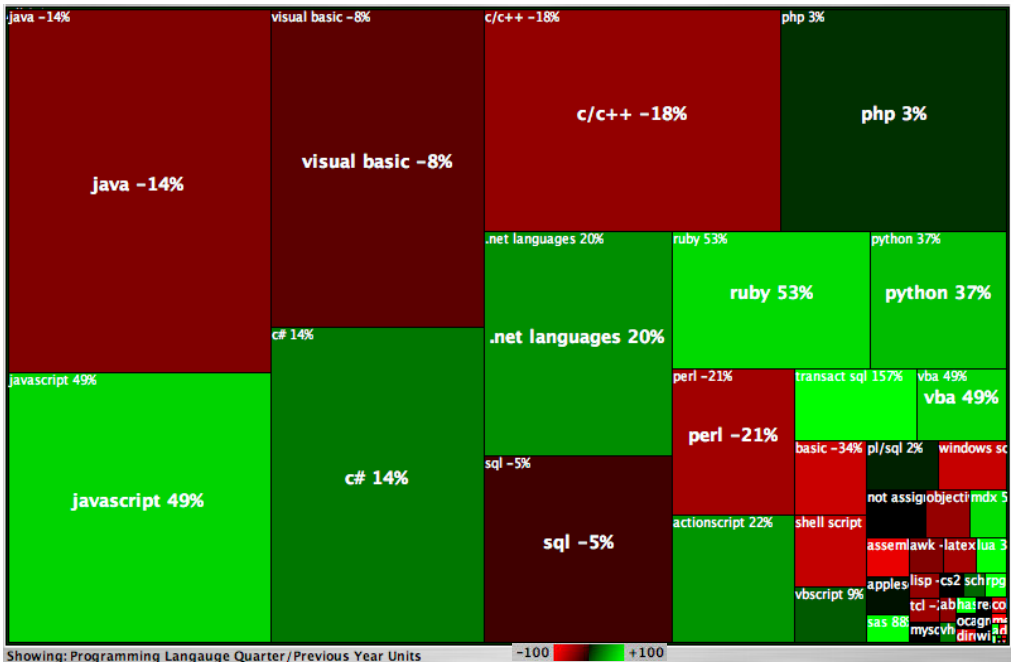
## REDHAT

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

## PROCTER & GAMBLE

Fluid dynamics simulation tools.

# Programming Language Book Market




Programming language book sales for Q4, 2007. Sizes of squares are relative to market size. Percentages displayed indicate growth from Q4, 2006. Green squares indicate a growing market. Red squares indicate a shrinking market. Used with permission from Oreilly Publishing: http://radar.oreilly.com/archives/2007/01/state_of_the_co_3.html



# Language Introduction

# 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 numpy 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
```

## SPLIT/JOIN STRINGS

```
# split space delimited words
>>> wrd_lst = s.split()
>>> print wrd_lst
['hello', 'world']

# join words back together
# with a space in-between
>>> ' '.join(wrd_lst)
hello world
```

# A few string methods

## REPLACING TEXT

```
>>> s = "hello world"
>>> s.replace('world','Mars')
'hello Mars'
```

## CONVERT TO UPPER CASE

```
>>> s.upper()
'HELLO MARS'
```

## REMOVE WHITESPACE

```
>>> s = "\t    hello    \n"
>>> s.strip()
'hello'
```



# Available string methods

```
# list available methods on a string
>>> dir(s)
[…
'capitalize',
'center',
'count',
'decode',
'encode',
'endswith',
'expandtabs',
'find',
'index',
'isalnum',
'isalpha',
'isdigit',
'islower',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'replace',
'rfind',
'rindex',
'rjust',
'rsplit',
'rstrip',
'split',
'splitlines',
'startswith',
'strip',
'swapcase',
'title',
'translate',
'upper',
'zfill']
```

# Multi-line Strings

```
# strings in triple quotes
# retain line breaks
>>> 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
```



# String Formatting

## 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
>>> t = "%s %f, %d" % (s,x,y)
>>> print t
some numbers: 1.340000, 2
```

For more information, see:
http://docs.python.org/lib/typesseq-strings.html
http://docs.python.org/lib/node40.html

More advanced templating engines such as "Cheetah" also exist:
http://www.cheetahtemplate.org/

## NAMED VARIABLES

```
# It also supports a 'named'
# variable approach where
# %(var_name)f will output
# var_name in float format.
>>> "%(s)s %(x)f, %(y)d" %
... locals()
>>> print t
some numbers: 1.340000, 2
```

## NAMED VARIABLES

```
# As of 2.4, Python supports
# "templates"
>>> from string import Template
>>> t = Template("$s $x $y")
>>> t.substitute(locals())
Some numbers: 1.34, 2
```

# 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

## 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
```

Prior to version 2.5, Python was limited to sequences with ~2 billion elements. Python 2.5 can handle up to $2^{63}$ elements.

## 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
True
>>> 13 not in l
False
```



# Slicing

**`var[lower:upper:step]`**

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). The step value specifies the stride between elements.

## 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]
# every other element
>>> l[::2]
[10, 12, 14]
```

# A few methods for list objects

**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.extend( sequence )**

Concatenate sequence onto this list.

**some_list.index( x )**

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

**some_list.insert( index, x )**

Insert x before the specified index

**some_list.pop( index )**

Return the element at the specified index. Also, remove it from 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

```
>>> 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
# extend with another list
>>> l.extend([5,4])
>>> print l
[10,21,23,11,24,11,5,4]
# where does 11 first occur?
>>> l.index(11)
3
# insert 100 at index 2?
>>> l.insert(2, 100)
>>> print l
[10,21,100,23,11,24,11,5,4]

# pop the item at index=4
>>> l.pop(3)
23
# remove the first 11
>>> l.remove(11)
>>> print l
[10,21,100,24,11,5,4]
# sort the list
>>> l.sort()
>>> print l
[4,5,10,11,21,24,100]
# reverse the list
>>> l.reverse()
>>> print l
[100,24,21,11,10,5,4]
```

# Assorted other list functions

## SORTED

```
# l.sort() is an inplace
# sort. sorted(l) returns a
# new list of the items in l
# sorted.
>>> l = [10,21,23,11,24]
>>> sorted(l)
[10, 11, 21, 23, 24]
```

## REVERSED

```
# reversed(l) returns an
# 'iterator' that will
# return elements of the
# list in reverse order.
# A copy is not made unless
# explicitly asked for.
>>> l = [10,21,23,11,24]
>>> for i in reversed(l):
...     print i,
24 11 23 21 10
```

## ZIP

```
# zip combines elements of
# multiple lists together as
# tuples.
>>> x = [1,2,3]
>>> y = ['a','b','c']
>>> z = zip(x, y)
>>> z
[(1,'a'), (2,'b'), (3, 'd')]

# zip (with a little trick) is
# also its own inverse
>>> zip(*z)
[(1, 2, 3), ('a', 'b', 'c')]
```



# 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

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

**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

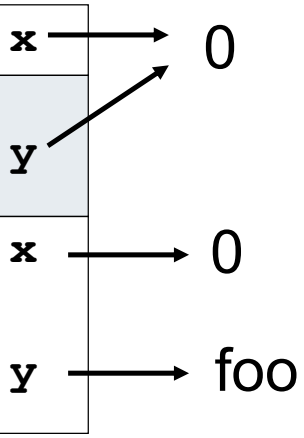
```
>>> d = {'cows': 1,'dogs':5,
...       'cats': 3}

# 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
{}
```



# Assignment of "simple" object

Assignment creates object references.

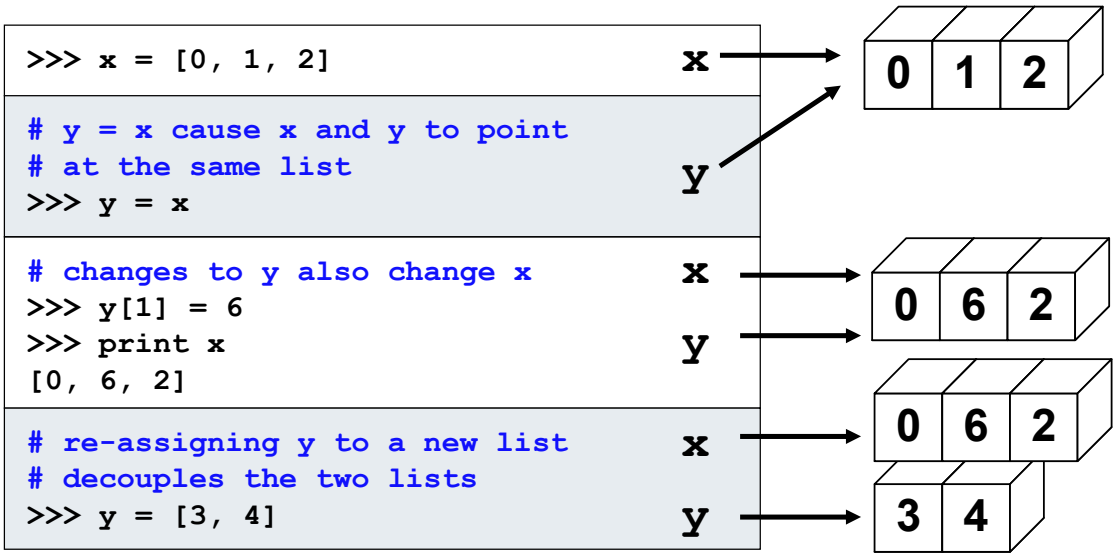
```
>>> x = 0

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

# re-assigning y to a new value
# decouples the two
>>> y = "foo"
>>> print x
0
```



# Assignment of Container object

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
```

It often pays to be explicit. If you are testing for an empty list, the test for:

```
if len(x) == 0:
    ...
```

This is more explanatory to future readers of your code. It also can avoid bugs where x==None my be passed in and unexpectedly go down this path.

# 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

```
# 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

**EX1.PY**

```
# 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**

```
# 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

**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*

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()
```

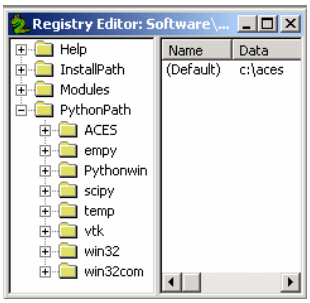


# Setting up PYTHONPATH

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/your_modules
```

### UNIX -- .bashrc

```
PYTHONPATH=$PYTHONPATH:$HOME/your
_modules
export PYTHONPATH
```

# Reading files

## 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 = l.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
```

See demo/reading_files directory for code.



# 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

## 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
```



# 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
```

# Pickling and Shelves

*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 particle
>>> some_particle = particle.Particle(2.0,3.0)
>>> f['my_favorite_particle'] = some_particle
>>> f.close()
< kill interpreter and restart! >
>>> import shelve
>>> f = shelve.open('c:/temp/pickle','r')
>>> some_particle = f['my_favorite_particle']
>>> some_particle.momentum()
6.0
```



# Exception Handling

## ERROR ON LOG OF ZERO

```
import math
>>> math.log10(10.0)
1.
>>> math.log10(0.0)
Traceback (innermost last):
OverflowError: math range error
```

## CATCHING ERROR AND CONTINUING

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

# Sorting

## 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
>>> from particle import Particle
>>> x = [Particle(1.2,3.4), Particle(2.1,2.3), Particle(4.6,.7)]
>>> sorted(x, cmp=by_mass)
[(m:1.2, v:3.4), (m:2.1, v:2.3), (m:4.6, v:0.7)]

>>> sorted(x, cmp=by_velocity)
[(m:4.6, v:0.7), (m:2.1, v:2.3), (m:1.2, v:3.4)]

>>> sorted(x, cmp=by_momentum)
[(m:4.6, v:0.7), (m:1.2, v:3.4), (m:2.1, v:2.3)]
```



See demo/particle directory for sample code

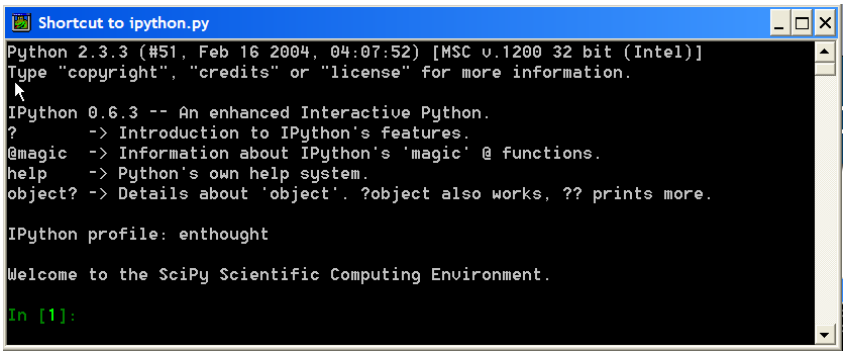
# IPython

An enhanced
interactive python shell



## IPython command prompt

- Available at http://ipython.scipy.org/
- Developed by Fernando Perez at University of Colorado at Boulder
- Provides a nice environment for scientific computing with Python



# IPython

## STANDARD PYTHON

```
In [1]: a=1

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

## 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
```

## AVAILABLE VARIABLES

```
In [6]: b = [1,2,3]

# List available variables.
In [7]: whos
Variable   Type    Data/Length
-----------------------------
a          int     1
b          list    [1, 2, 3]
```



# Shell Commands

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

# list directory contents
In [10]: ls
 Volume in drive C has no label.
 Volume Serial Number is E0B4-1D2D

 Directory of c:\demo\speed_of_light
11/11/2004  03:51 PM <DIR>          .
11/11/2004  03:51 PM <DIR>          ..
11/08/2004  11:45 PM             1,188 exercise_speed_of_light.txt
11/08/2004  10:52 PM         2,682,023 measurement_description.pdf
11/08/2004  10:44 PM           187,087 newcomb_experiment.pdf
11/08/2004  10:51 PM             1,402 speed_of_light.dat
11/11/2004  03:51 PM             1,017 speed_of_light.py
8 File(s)      2,874,867 bytes
               2 Dir(s)   8,324,673,536 bytes free
```

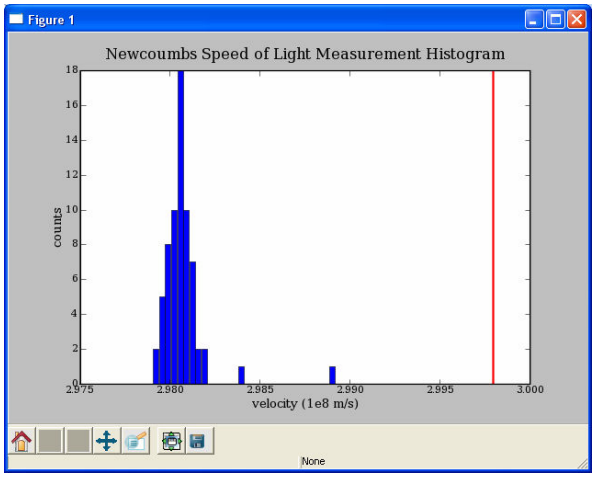
# Directory History

```
# change directory
In [1]: cd c:/demo/speed_of_light
c:\demo\speed_of_light

# list the history of directory changes
In [2]: dhist
Directory history (kept in _dh)
0: C:\Python23\Scripts
1: c:\demo\speed_of_light

# change to a directory in the history chosen by number.
In [3]: cd -0
C:\Python23\Scripts
```



# Magic commands

```
# 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
```



# Magic Commands

```
# pdef prints the definition for a command
In [45]: pdef stats.histogram
stats.histogram(a, numbins=10, defaultlimits=None, printextras=1)

# psource prints the source code for a command
In [43]: psource squeeze
def squeeze(a):
    "Returns a with any ones from the shape of a removed"
    a = asarray(a)
    b = asarray(a.shape)
    val = reshape (a, tuple (compress (not_equal (b, 1), b)))
    return val
```

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



# Magic Commands

```
# ? prints information about an object
In [46]: stats.histogram?
Type:        function
Base Class:  <type 'function'>
String Form: <function histogram at 0x02289730>
Namespace:   Interactive
File:        c:\python23\lib\site-packages\scipy\stats\stats.py
Definition:  stats.histogram(a, numbins=10, defaultlimits=None, printextras=1)
Docstring:
    Returns (i) an array of histogram bin counts, (ii) the smallest value
of the histogram binning, and (iii) the bin width (the last 2 are not
necessarily integers).  Default number of bins is 10.  Defaultlimits
can be None (the routine picks bins spanning all the numbers in the
a) or a 2-sequence (lowerlimit, upperlimit).  Returns all of the
following: array of bin values, lowerreallimit, binsize, extrapoints.

Returns: (array of bin counts, bin-minimum, min-width, #-points-outside-range)
```

# Selections from the Standard Library



## FTP -- Sending binary files

```
# Generate a plot
>>> plot((1,2,3))
>>> savefig('simple_plot.png') # broken in current release

# Open an ftp connection
>>> import ftplib
>>> server = 'www.enthought.com'
>>> user,password = 'eric', 'python'
>>> site = ftplib.FTP(server, user, password)

# Change to web directory on server
>>> homepage_folder = 'public_html'
>>> site.cwd(homepage_folder)

# Send file to site in binary format, and clean up
>>> img = open('simple_plot.png')
>>> site.storbinary('STOR plot.png', img)
>>> img.close()
>>> site.quit()
```

# FTP – Sending text

```
# Generate html page for plot
>>> html = '''<h1> A Nice Plot </h1>
              <img src="plot.png">
            '''

# Make a file-like object from string
>>> import cStringIO
>>> html_file = cStringIO.StringIO(html)

# Open an ftp connection (variables from previous page)
>>> import ftplib
>>> site = ftplib.FTP(server, user, password)
>>> site.cwd(homepage_folder)
>>> site.storlines('STOR plot.html', html_file)
>>> img.close()
>>> site.quit()
```



# FTP -- Retrieving files

```
# Open an ftp connection (variables from previous page)
>>> import ftplib
>>> site = ftplib.FTP(server, user, password)
>>> site.cwd(homepage_folder)

# Grab directory contents
>>> site.retrlines('LIST')
total 12
-rw-r--r--    1 eric  eric       56 Aug 26 14:41 plot.html
-rw-r--r--    1 eric  eric     5111 Aug 26 14:38 plot.png
'226 Transfer complete.'

# Grab a file and stick data in a string
>>> import cStringIO
>>> data_buffer = cStringIO.StringIO()
>>> site.retrlines('RETR plot.html',data_buffer.write)
'226 Transfer complete.'
>>> data_buffer.getvalue()
'<h1>Nice Plot</h1>       <img src="plot.png">'
```

# Browsers and HTTP

## LAUNCH AN EXTERNAL WEB BROWSER

```
>>> import webbrowser
>>> webbrowser.open('http://www.enthought.com/~eric/plot.html')
<creates a web browser on your platform for viewing>
```

## USING `httplib`

```
>>> import httplib
>>> site = httplib.HTTP('http://www.enthought.com')
# setup a request header (RFC822 format)
>>> site.putrequest('GET','/~eric/plot.html')
>>> site.putheader('Accept','text/html')
>>> site.putheader('Accept','text/plain')
>>> site.endheaders()
# Retrieve the file and print the data
>>> errcode,errmsg,headers = site.getreply()
>>> file = site.getfile()
>>> file.read()
'<h1>Nice Plot</h1>       <img src="plot.png">'
>>> file.close()
```



# Telnet –Remote system control

```
>>> import telnetlib
>>> tn = telnetlib.Telnet('some.host.com')
# Read text up to the "login: " prompt and send login name
>>> tn.read_until("login: ")
< outputs some text >
>>> tn.write("ej\n")
# Read text up to the password prompt and send password
>>> tn.read_until("Password: ")
"ej's Password: "
>>> tn.write(secret_password + "\n")
# Retrieve directory listing and exit.
>>> tn.write("ls\n")
>>> tn.write("exit\n")
# Read all text from telnet session since password prompt.
>>> results  = tn.read_all()
>>> print results
< prints the login banner message and other stuff >
> ls
foo.c        info          bin
> exit   logout
```

# Encryption

## RFC 1321 CHECKSUM -- md5

```
# md5 creates a 128 bit "fingerprint" of an input string.
>>> import md5
>>> sumgen = md5.new()
>>> sumgen.update("A simple text string.")
# digest returns the raw 16 byte string
>>> sumgen.digest()
'\xf2\x02\xd6!\xdb\xd5\xcb\xe1Y\xca\xdd\xf4\xe3\x1cp\xb5'
# or as a human readable 32 byte string of hex digits
>>> sumgen.hexdigest()
'f202d621dbd5cbe159caddf4e31c70b5'
```

## PASSWORD ENCRYPTION (UNIX ONLY) -- crypt

```
>>> import crypt
# 1st argument is password to encode, 2nd is 2 character 'salt'
>>> crypt.crypt('passwd','aa')
aaU3oayJ5BcR6
```



# Packing/Unpacking C structures

The **struct** module can pack/unpack multiple values into/from a binary structure represented as a Python string. The memory layout of the structure is specified using a format string. This is often useful for reading binary files or communicating with C functions. See standard Python reference for formatting options.

```
>>> import struct
# pack values as int, int, & signed byte, or 'iib'  format
>>> struct.pack('iib',15,2,3)
'\x0f\x00\x00\x00\x02\x00\x00\x00\x03'
# Force byte ordering to big-endian using '>'
>>> struct.pack('>iib',15,2,3)
'\x00\x00\x00\x0f\x00\x00\x00\x02\x03'
# Determine the number of bytes required by a structure
>>> struct.calcsize('iib')
9
>>> struct.unpack('iib','\x0f\x00\x00\x00\x02\x00\x00\x00\x03')
(15, 2, 3)
```

# Remote Call example -- xmlrpc

## XMLRPC FACTORIAL SERVER

```
import SocketServer, xmlrpcserver
def fact(n):
    if n <= 1:    return 1
    else:         return n * fact(n-1)

# Override the call() handler to call the requested function
class my_handler(xmlrpcserver.RequestHandler):
    def call(self, method, params):
        print "CALL", method
        return apply(eval(method),params)

# Start a server listening for request on port 8001
if __name__ == '__main__':
    server = SocketServer.TCPServer(('', 8001), my_handler)
    server.serve_forever()
```

## CLIENT CODE -- CALLS REMOTE FACTORIAL FUNCTION

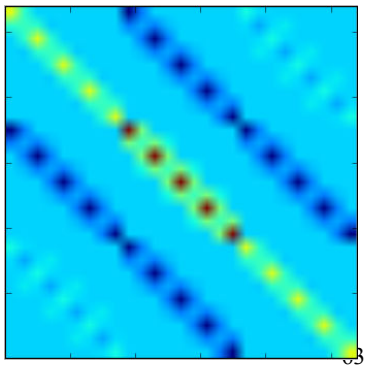
```
>>> import xmlrpclib
>>> svr = xmlrpclib.Server("http://localhost:8001")
>>> svr.fact(10)
3628800
```



# Retrieving remote files

## GRAB AND VIEW MATRIX FROM THE MATRIX MARKET SERVER

```
import gzip
import os
from tempfile import mktemp
from urllib import urlretrieve
from scipy import io
from matplotlib.pylab import imshow, show
url="ftp://math.nist.gov/pub/MatrixMarket2/SPARSKIT/fidap/fidap005.mtx.gz"
fname =  mktemp(".mtx.gz")
print "Downloading Matrix Market; this may take a minute..."
urlretrieve(url, fname)
a = io.mmread(gzip.open(fname))
imshow(a.toarray())
show()
os.unlink(fname)
```



Source available in demo/url_retreive/matrix_market.py

# Pipes

```
>>> import os
# Change directories
>>> os.chdir('c:\\')
# Get command output
# from pipe.
>>> p = os.popen('dir')
>>> print p.read()
Volume C has no label.
Volume Serial is 58C7-F5CD

 Directory of C:
07/10/01  4:57p <DIR>  act
08/03/01  3:38p <DIR>  ATI
...
```

```
# Use popen2 to get pipes for both
# reading and writing.
>>> snd,rcv = os.popen2('grep dog')
>>> snd.write('1 cat\n 2 dog\n')
>>> snd.close()
>>> print rcv.read()
 2 dog
```

Take care when using input and output pipes. Most OSes buffer IO which can lead to unexpected behavior. When sending data through a pipe, call .flush() or .close() to force the write. When reading data, your at the mercy of the other process and the OS. rcv.read() can deadlock in worst case situations.



# Interactive help

# >>> help()

- The help() builtin function provides access to help/documentation text for various modules, classes, functions, commands, objects, etc.
- Prints a “man page” to stdout
- Scans modules, classes, etc. and looks at docstrings in code in order to create help text



# >>> help()

**HELP FROM INTERPRETER**

```
# Passing help() a string will look it up in modules etc.
>>> help("numpy")
Help on package numpy:

NAME
    numpy

FILE
    c:\python24\lib\site-packages\numpy-1.0.2 ... .-win32.egg\numpy\__init__.py

DESCRIPTION
    NumPy
    ==========

    You can support the development of NumPy and SciPy by purchasing
    the book "Guide to NumPy" at

      http://www.trelgol.com

    It is being distributed for a fee for only a few years to
    cover some of the costs of development.  After the restriction period
    it will also be freely available.
<snip>
```

# >>> help()

## HELP ON OBJECT

```
# Passing help() an object also works.
>>> from numpy import array
>>> help(array)
In [7]: help(array)
Help on built-in function array in module numpy.core.multiarray:

array(...)
    array(object, dtype=None, copy=1,order=None, subok=0,ndmin=0)

    Return an array from object with the specified date-type.

    Inputs:
      object - an array, any object exposing the array interface, any
               object whose __array__ method returns an array, or any
               (nested) sequence.
      dtype  - The desired data-type for the array.  If not given, then
               the type will be determined as the minimum type required
               to hold the objects in the sequence.  This argument can only
               be used to 'upcast' the array.  For downcasting, use the
               .astype(t) method.
<snip>
```



# Documenting your code

## TYPICAL DOCUMENTATION FORMAT

```
# foo.py
""" Short docstring for the module goes here.

    A longer description for the module goes it here.  It
    is typically multiple lines long.
"""
class Foo:
    """ Short docstring for Foo class.

        Longer docstring describing the Foo class.
    """
    def some_method(self):
        """ Short description for some_method.

            Longer description for some_method...
        """

def bar():
    """ Short docstring for bar method.

        And, suprisingly, the long description for the
        method.
    """
```

# Documenting your code

## HELP() OUTPUT FOR MODULE

```
In [12]: import foo
In [14]: help(foo)
Help on module foo:

NAME
    foo - Short docstring for the module goes here.
FILE
    c:\eric\my documents\presentations\python_class\demo\docstrings\foo.py
DESCRIPTION
    A longer description for the module goes it here.  It
    is typically multiple lines long.
CLASSES
    Foo
    class Foo
     |  Short docstring for Foo class.
     |
     |  Longer docstring describing the Foo class.
     |
     |  Methods defined here:
     |
     |  some_method(self)
     |      short description for some_method.
     |
<snip>
```



# pydoc

- Pydoc allows for access to help text via keyword (like “man -k”), or through a web browser.

- Pydoc can generate html automatically

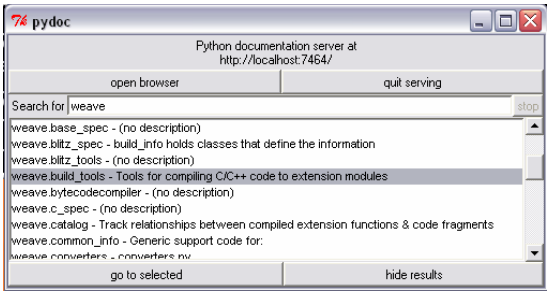
# pydoc

## CREATING HTML DOCS WITH PYDOC

```
C:\>pydoc -w foo

wrote foo.html
```


## GRAPHICAL DOCUMENTATION BROWSER

```
C:\>pydoc -g
```






# Ipython help

## IPYTHON'S '?' RETURNS HELP FOR OBJECTS

```
In [22]: from numpy import array

In [31]: array?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in function array>
Namespace:      Interactive
Docstring:
    array(object, dtype=None, copy=1,order=None, subok=0,ndmin=0)

Return an array from object with the specified date-type.

Inputs:
  object - an array, any object exposing the array interface, any
           object whose __array__ method returns an array, or any
           (nested) sequence.
  dtype  - The desired data-type for the array.  If not given, then
           the type will be determined as the minimum type required
           to hold the objects in the sequence.  This argument can only
           be used to 'upcast' the array.  For downcasting, use the
<snip>
```

# Ipython help

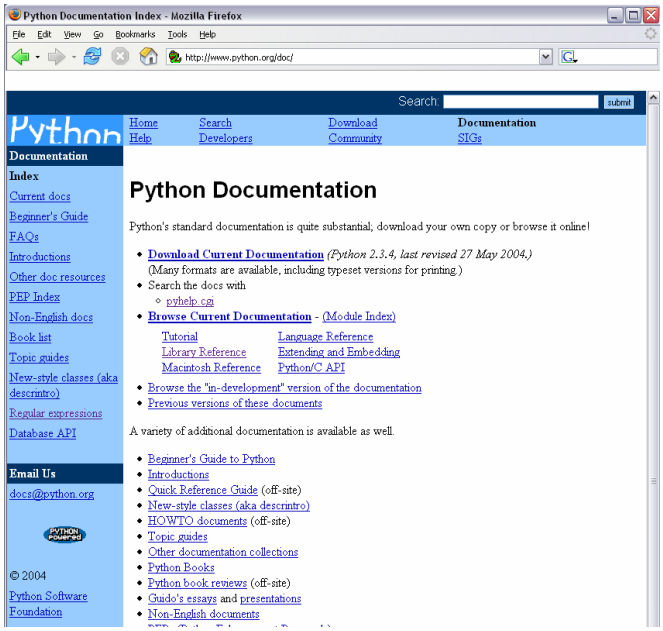
## Add a question mark to any object for access to it's help information

```
In [24]: a = array([[1,2,3],[4,5,6]])

In [25]: a?
Type:           array
String Form:
[[1 2 3]
 [4 5 6]]
Namespace:      Interactive
Length:         2
Docstring:
    A array object represents a multidimensional, homogeneous array of basic
values.  It has the folowing data members, m.shape (the size of each dimension
in the array), m.itemsize (the size (in bytes) of each element of the array),
and m.typecode (a character representing the type of the matrices elements).
Matrices are sequence, mapping and numeric objects.  Sequence indexing is
similar to lists, with single indices returning a reference that points to the
old matrices data, and slices returning by copy.  A array is also allowed to
be indexed by a sequence of items.  Each member of the sequence indexes the
corresponding dimension of the array. Numeric operations operate on matrices
in an element-wise fashion.
```



# Another excellent source of help...

## http://www.python.org/doc



ENTHOUGHT

# Numpy



ENTHOUGHT

## Numpy

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

- Offers Matlab-ish capabilities within Python
- Numpy replaces Numeric and Numarray
- Developed by Travis Oliphant
- 27 svn “committers” to the project
- Numpy 1.0 released October, 2006
- ~16K downloads/month from Sourceforge.

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

# Getting Started

## IMPORT NUMPY

```
>>> from numpy import *
>>> __version__
1.0.2.dev3487
```

or

```
>>> 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.

While IPython is used for all the demos, '>>>' is used on future slides instead of '`In [1]:`' because it takes up less room.



# Array Operations

## SIMPLE ARRAY MATH

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

Numpy defines the following constants:

pi = 3.14159265359

e = 2.71828182846

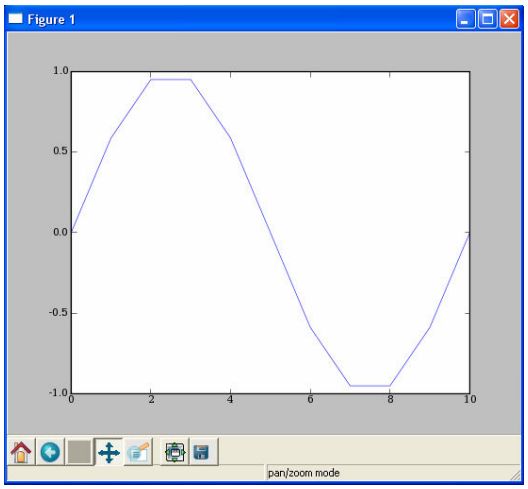
## MATH FUNCTIONS

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

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

# inplace operations
>>> x *= a
>>> x
array([ 0.,0.628,…,6.283])

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

# Plotting Arrays

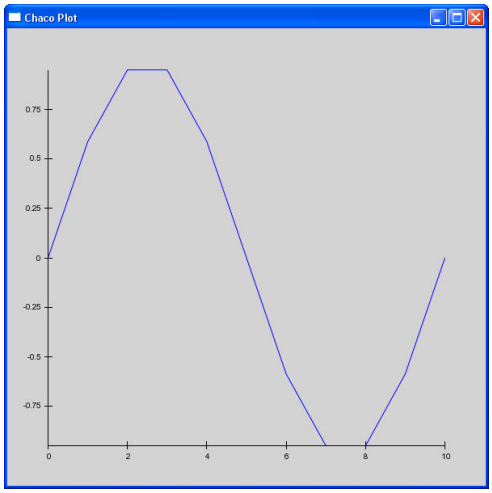
## MATPLOTLIB

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




## CHACO SHELL

```
>>> from enthought.chaco2 \
...     import shell
>>> shell.plot(x,y)
```






# Introducing Numpy Arrays

## 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.dtype
dtype('int32')
```

## BYTES PER ELEMENT

```
>>> a.itemsize # per element
4
```

## ARRAY SHAPE

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

## ARRAY SIZE

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

# Introducing Numpy Arrays

ENTHOUGHT

## BYTES OF MEMORY USED

```
# returns the number of bytes
# used by the data portion of
# the array.
 >>> a.nbytes
12
```

## NUMBER OF DIMENSIONS

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

## ARRAY COPY

```
# create a copy of the array
>>> b = a.copy()
>>> b
array([0, 1, 2, 3])
```

## CONVERSION TO LIST

```
# convert a numpy array to a
# python list.
>>> a.tolist()
[0, 1, 2, 3]

# For 1D arrays, list also
# works equivalently, but
# is slower.
>>> list(a)
[0, 1, 2, 3]
```



# Setting Array Elements

ENTHOUGHT

## ARRAY INDEXING

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

## FILL

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

# This also works, but may
# be slower.
>>> a[:] = 1
>>> a
[1, 1, 1, 1]
```

## BEWARE OF TYPE COERSION

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

# assigning a float to into
# an int32 array will
# truncate decimal part.
>>> a[0] = 10.6
>>> a
[10, 1, 2, 3]

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

# Multi-Dimensional Arrays

ENTHOUGHT

## 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)

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

## ELEMENT COUNT

```
>>> a.size
8
>>> size(a)
8
```

## NUMBER OF DIMENSIONS

```
>>> a.ndims
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 FIRST ROW USING SINGLE INDEX

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



# Array Slicing

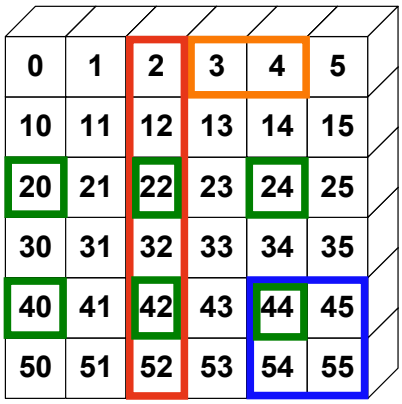
ENTHOUGHT

## SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

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

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

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

## STRIDES ARE ALSO POSSIBLE

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



# Slices Are References

Slices are references to memory in 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[0] = 10

# changing b changed a!
>>> a
array([ 1,  2, 10, 3, 4])
```



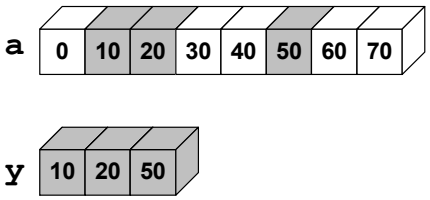
# Fancy Indexing

## INDEXING BY POSITION

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


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

# using take
>>> y = take(a,[1,2,-3])
>>> print y
[10 20 50]
```

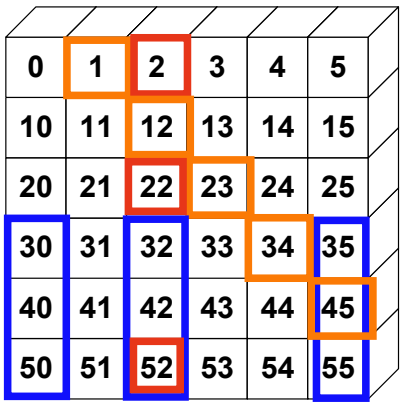
## INDEXING WITH BOOLEANS

```
>>> mask = array([0,1,1,0,0,1,0,0],
...              dtype=bool)

# fancy indexing
>>> y = a[mask]
>>> print y
[10,20,50]

# using compress
>>> y = compress(mask, a)
>>> print y
[10,20,50]
```



# Fancy Indexing in 2D

```
>>> 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])
```

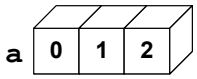



Unlike slicing, fancy indexing creates copies instead of views into original arrays.



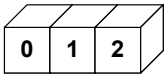
# Indexing with None

`None` is a special index that inserts a new axis in the array at the specified location. Each `None` increases the arrays dimensionality by 1.



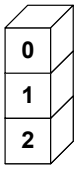

### 1 X 3

```
>>> y = a[None,:]
>>> shape(y)
(1, 3)
```



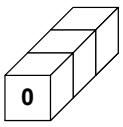

### 3 X 1

```
>>> y = a[:,None]
>>> shape(y)
(3, 1)
```




### 3 X 1 X 1

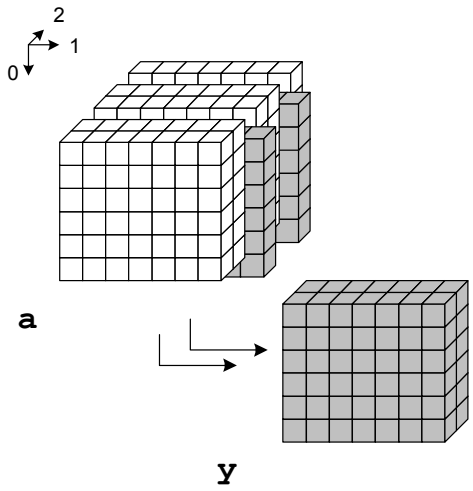
```
>>> y = a[:,None, None]
>>> shape(y)
(3, 1, 1)
```



# 3D Example

## MULTIDIMENSIONAL

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

# The take() function also works.
>>> y = take(a,[2,-2], axis=2)
```






# "Flattening" Arrays

## a.flatten()

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

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

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

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

no change

## a.flat

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

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

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

changed!

# "(Un)raveling" Arrays

## a.ravel()

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

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

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

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

changed!

## a.ravel()

```
# Create a 2D 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 will create 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 AND RESHAPE

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


# reshape array inplace 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
```

# Re-ordering Dimensions

## TRANSPOSE

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



# 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.squeeze()
>>> a.shape
(2,3)
```

# Diagonals

ENTHOUGHT

## DIAGONAL

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


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

# Use offset to move off the
# main diagonal.
>>> 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
>>> i = array([0,1])
# upper diagonal
>>> a[i,i+1] = 1
# lower diagonal
>>> a[i+1,i]= = -1
>>> a
array([[ 2,  1, 13],
       [-1,  2,  1],
       [31, -1,  2]])
```



# Complex Numbers

ENTHOUGHT

## COMPLEX ARRAY ATTRIBUTES

```
>>> a = array([1+1j,1,2,3])
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([0.-1.j, 1.-2.j, 2.-3.j,
       3.-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.,2,3])
>>> a.dtype
dtype('float64')
>>> a.nbytes
32
```

notice decimal

### 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('unit8')
>>> a.nbytes
4
```

### ARRAYS REFERENCING SAME DATA

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



# 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 int in C for the platform |
| Unsigned Integer | uint8, uint16, uint32, uint64, uint128, uint | uint defaults to the size of unsigned int in C for the platform |
| Float | float32, float64, float | Float is always a double precision floating point value (64 bits). |
| Complex | complex64, complex128, complex | The real and complex elements of a complex64 are each represented by a single precision (32 bit) value for a total size of 64 bits. |
| Strings | str, unicode | |
| Object | object | |

# Type Casting

ENTHOUGHT

## ASARRAY

```
>>> a = array((1.2, -3),
...           dtype=float32)
>>> a
array([ 1.20000005, -3.    ],
      dtype=float32)
# upcast
>>> asarray(a, dtype=float64)
array([ 1.20000005,-3.])

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

## ASTYPE

```
>>> a = array((1.2,-3))

>>> a.astype(float32)
array([ 1.20000005, -3.    ],
      dtype=float32)

>>> a.astype(uint8)
array([  1, 253],dtype=unit8)
```



# Array Calculation Methods

ENTHOUGHT

## SUM FUNCTION

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

# like add.reduce(), sum
# defaults to reducing
# along the first axis
# For 2D, this sums up columns.
>>> sum(a)
array([5., 7., 9.])

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

# Use flat to sum all values
>>> sum(a.flat)
21.
```

## SUM ARRAY METHOD

```
# The a.sum() defaults to
# summing *all* array values
>>> a.sum()
21.

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

## PRODUCT

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

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

# Min/Max

## MIN

```
>>> a = array([2.,3.,0.,1.])
>>> a.min(a, axis=0)
0.
# use Numpy's amin() instead
# of Python's builtin min()
# for speed operations on
# multi-dimensional arrays.
>>> amin(a, axis=0)
0.
```

## MAX

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

# functional form
>>> amax(a, axis=0)
3.
```

## 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]], float)

# 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]], float)

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

## POINT TO POINT

```
# Calculate max - min for
# array along columns
>>> a.ptp(axis=0)
array([ 3.0,  3.0,  3.0])
# max - min for entire array.
>>> a.ptp(axis=None)
5.0
```



# Summary of (most) array attributes/methods

## BASIC ATTRIBUTES

```
a.dtype - Numerical type of array elements. float32, uint8, etc.
a.shape - Shape of the array. (m,n,0,...)
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 is 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) - Change the size/shape of an array in-place.
a.swapaxes(axis1, axis2) - Swap the order of two axes in an array.
a.transpose(*axes) - Swap the order of any number of array axes.
a.T - Shorthand for a.transpose()
a.squeeze() - Remove any length=1 dimensions from an array.
```

# Summary of (most) array attributes/methods

ENTHOUGHT

## FILL AND COPY

```
a.copy() - Return a copy of the array.
a.fill(value) - Fill array with a scalar value.
```

## CONVERSION / COERSION

```
a.tolist() - Convert array into nested lists of values.
a.tostring() - raw copy of array memory into a python string.
a.astype(dtype) - Return array coerced to given dtype.
a.byteswap(False) - Convert byte order (big <-> little endian).
```

## COMPLEX NUMBERS

```
a.real - Return the real part of the array.
a.Imag - Return the imaginary part of the array.
a.conjugate() - Return the complex conjugate of the array.
a.conj()- Return the complex conjugate of an array.(same as conjugate)
```



# Summary of (most) array attributes/methods

ENTHOUGHT

## SAVING

```
a.dump(file) - Store a binary array data out to the given file.
a.dumps() - returns the binary pickle of the array as a string.
a.tofile(fid, sep="", format="%s") Formatted ascii output to file.
```

## SEARCH / SORT

```
a.nonzero() - Return indices for all non-zero elements in a.
a.sort(axis=-1) - Inplace sort of array elements along axis.
a.argsort(axis=-1) - Return indices for element sort order along axis.
a.searchsorted(b) - Return index where elements from b would go in a.
```

## ELEMENT MATH OPERATIONS

```
a.clip(low, high) - Limit values in array to the specified range.
a.round(decimals=0) - Round 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

## 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) - Sum up values along axis.
a.prod(axis=None) - Find the product of all values along axis.
a.min(axis=None)- Find the minimum value along axis.
a.max(axis=None) - Find the maximum value along axis.
a.argmin(axis=None) - Find the index of the minimum value along axis.
a.argmax(axis=None) - Find the index of the maximum value along axis.
a.ptp(axis=None) - Calculate a.max(axis) - a.min(axis)
a.mean(axis=None) - Find the mean (average) value along axis.
a.std(axis=None) - Find the standard deviation along axis.
a.var(axis=None) - Find the variance along axis.

a.any(axis=None) - True if any value along axis is non-zero.   (or)
a.all(axis=None) - True if all values along axis are non-zero. (and)
```



# Array Creation Functions

**`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. 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,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),typecode=float32)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]],dtype=float32)
```

# Array Creation Functions (cont.)

ENTHOUGHT

## IDENTITY

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

## EMPTY AND FILL

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

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

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



# Array Creation Functions (cont.)

ENTHOUGHT

## LINSPACE

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

## LOGSPACE

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

## 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])
```

# Array Creation Functions (cont.)

## MGRID

```
# get equally spaced point
# 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.]])
```

## BMAT

```
# 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)
```

### `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
```

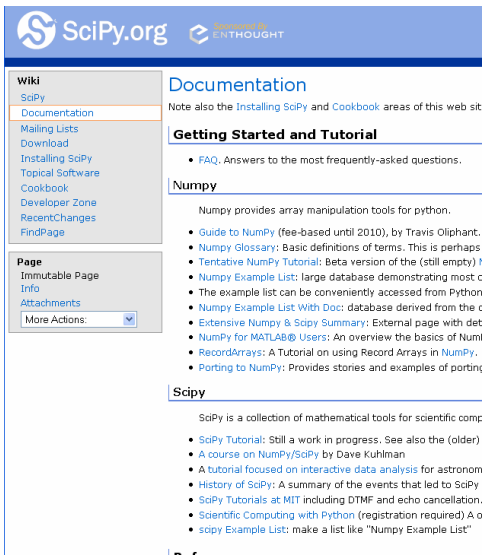
## OTHER USEFUL FUNCTIONS

```
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
```

# Helpful Sites

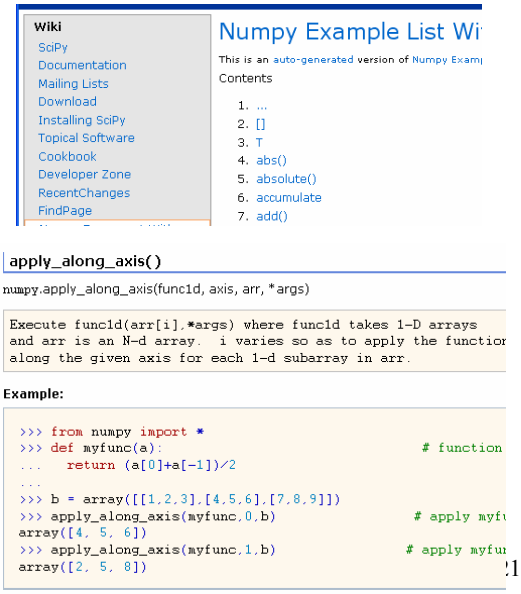
## SCIPY DOCUMENTATION PAGE

http://www.scipy.org/Documentation




## NUMPY EXAMPLES

http://www.scipy.org/Numpy_Example_List_With_Doc




# 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])
```

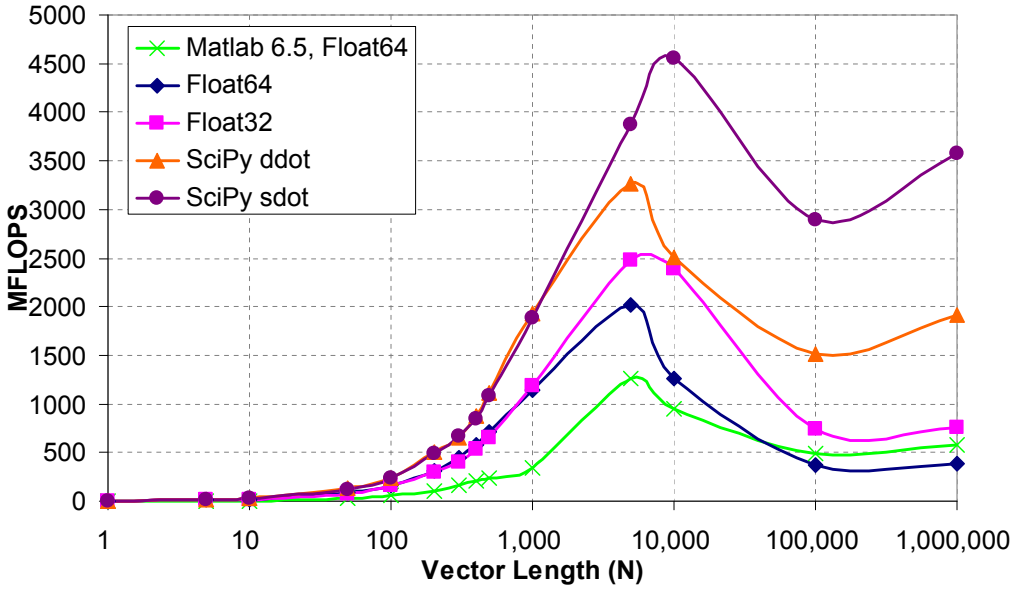
## ADDITION USING AN OPERATOR FUNCTION

```
>>> 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])
```

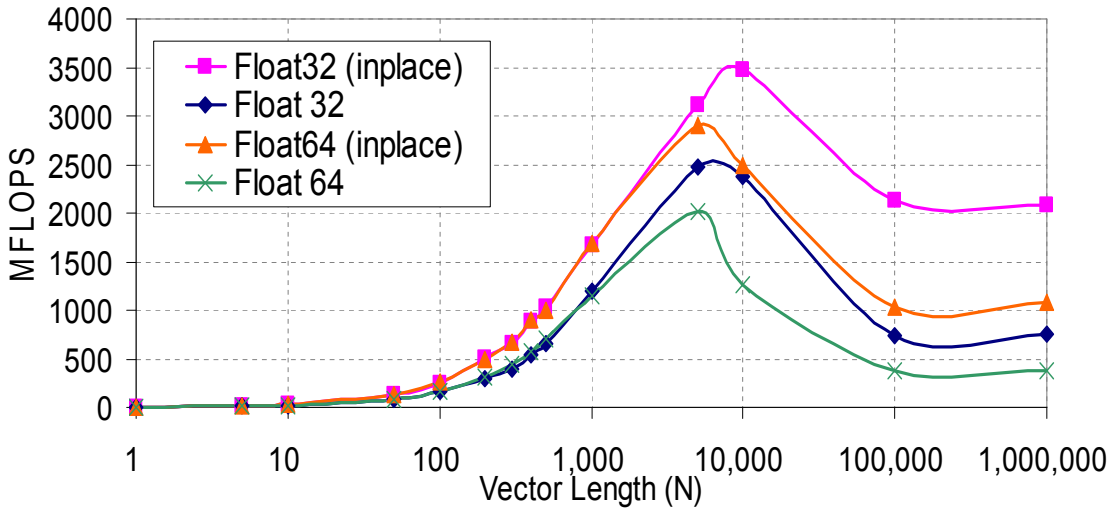
# Vector Multiply Speed




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




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

**Your mileage may vary.**

# 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

| | | | | |
|---|---|---|---|---|
| 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)
```

# 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:

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

**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**

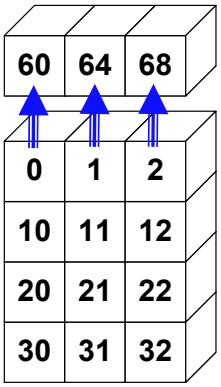
```
>>> a = array([1,1,0,1])
>>> logical_and.reduce(a)
0
>>> logical_or.reduce(a)
1
```

# op.reduce()

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.

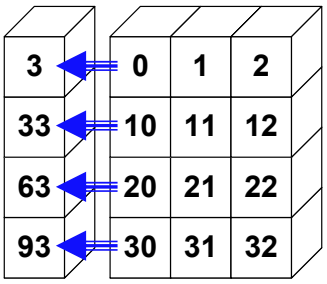
## SUM COLUMNS BY DEFAULT

```
>>> 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 = 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()

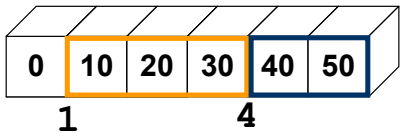
`op.reduceat(a,indices)` applies `op` to ranges in the 1d array `a` defined by the values in `indices`. The resulting array has the same length as `indices`.

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

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

**EXAMPLE**

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

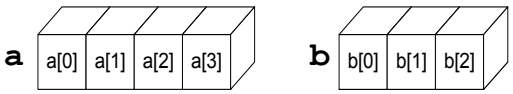



For multidimensional arrays, `reduceat()` is always applied along the *last* axis (sum of rows for 2D arrays). This is inconsistent with the default for `reduce()` and `accumulate()`.



# 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)
```

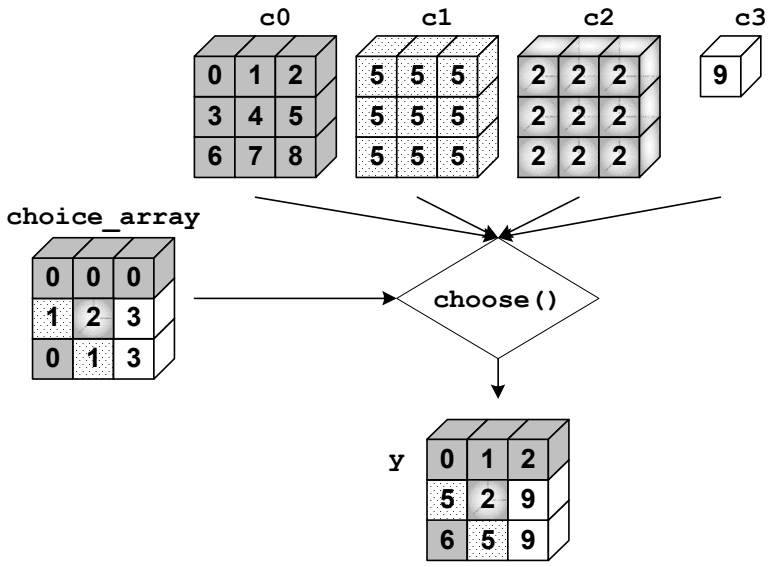
| | | |
|---|---|---|
| a[0]+b[0] | a[0]+b[1] | a[0]+b[2] |
| a[1]+b[0] | a[1]+b[1] | a[1]+b[2] |
| a[2]+b[0] | a[2]+b[1] | a[2]+b[2] |
| a[3]+b[0] | a[3]+b[1] | a[3]+b[2] |

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

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

# Array Functions – choose()

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






# Example - choose()

## CLIP LOWER VALUES TO 10

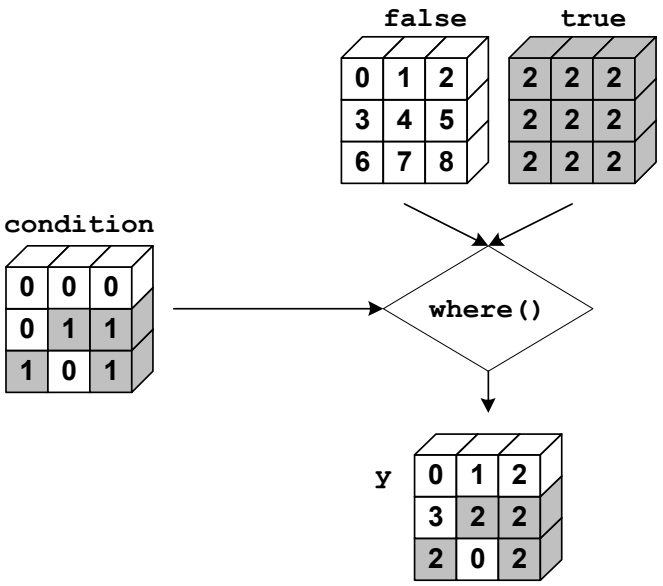
```
>>> a
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])

>>> 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)
```

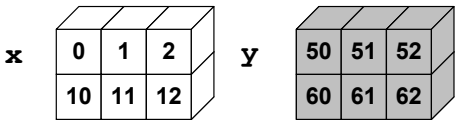




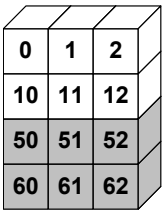

# Array Functions – `concatenate()`

**`concatenate((a0,a1,…,aN),axis=0)`**

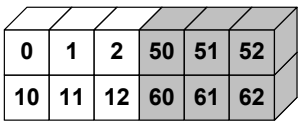
The input arrays `(a0,a1,…,aN)` will be concatenated along the given `axis`. They must have the same shape along every axis *except* the one given.



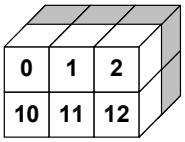

```
>>> concatenate((x,y))
```



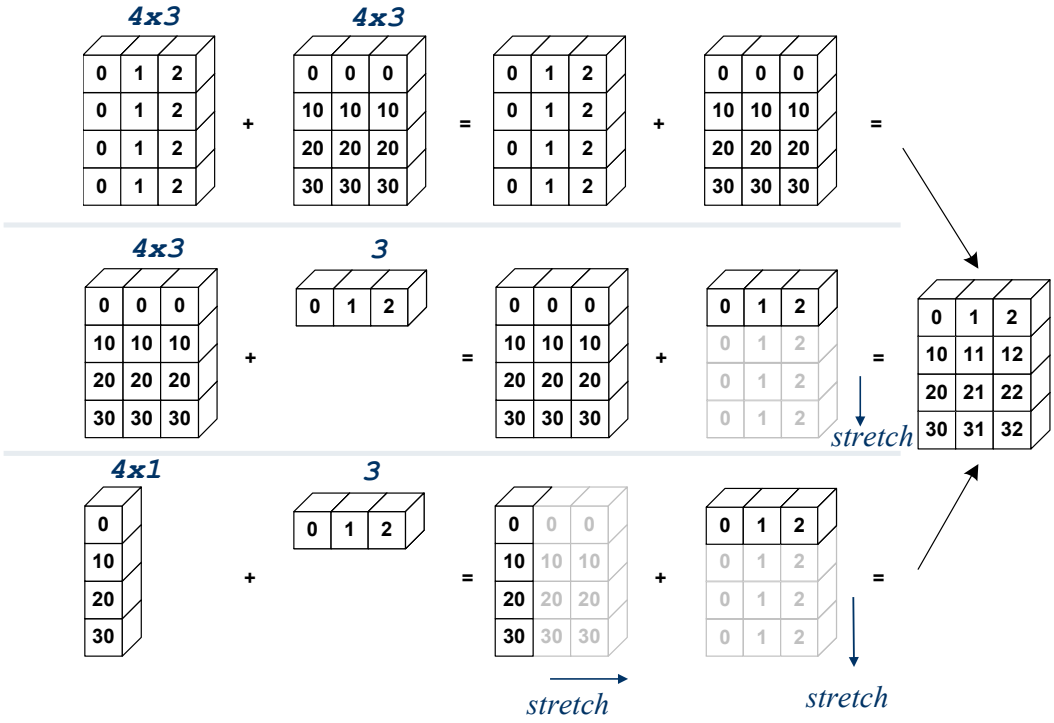

```
>>> concatenate((x,y),1)
```




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



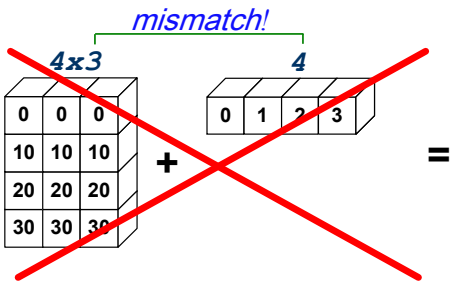
# Array Broadcasting





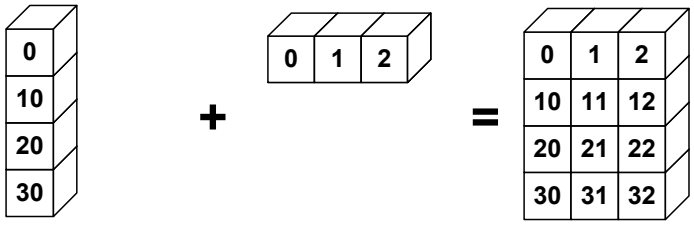

# Broadcasting Rules

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.



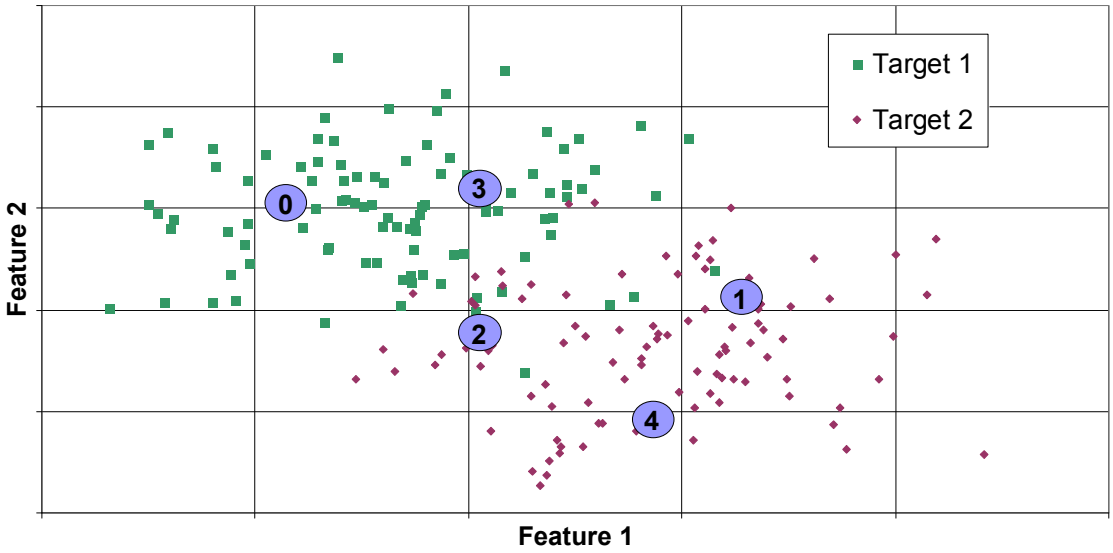
# Broadcasting in Action

```
>>> a = array((0,10,20,30))
>>> b = array((0,1,2))
>>> y = a[:, None] + b
```

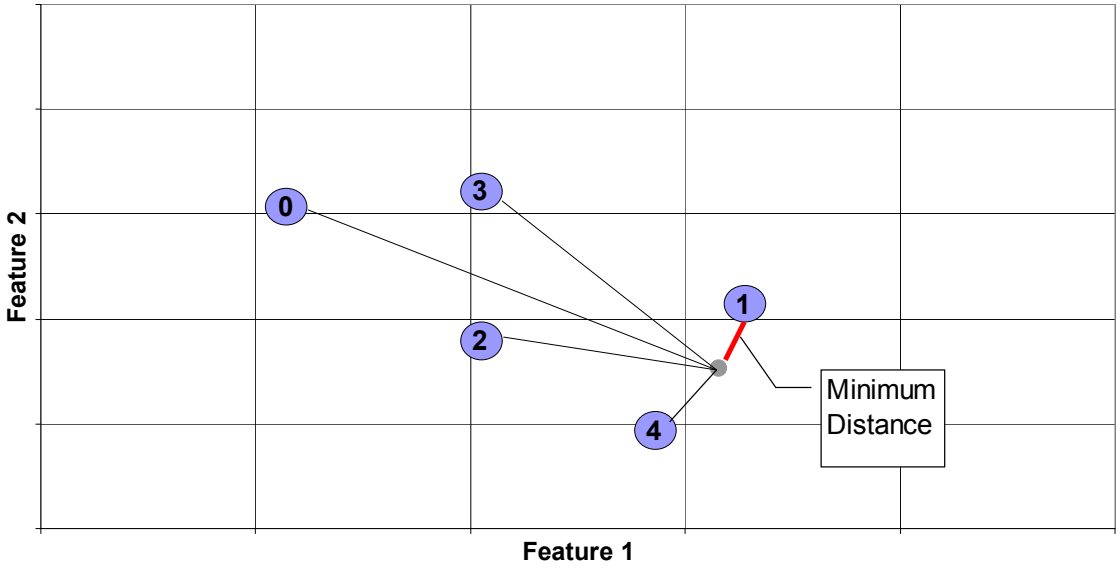





# Vector Quantization Example

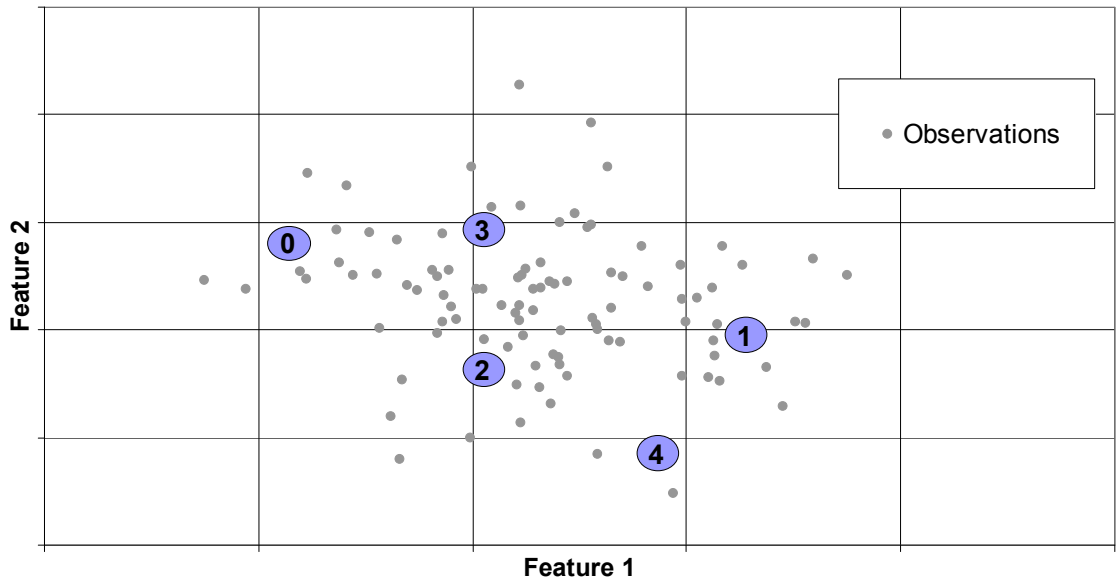


# Vector Quantization Example

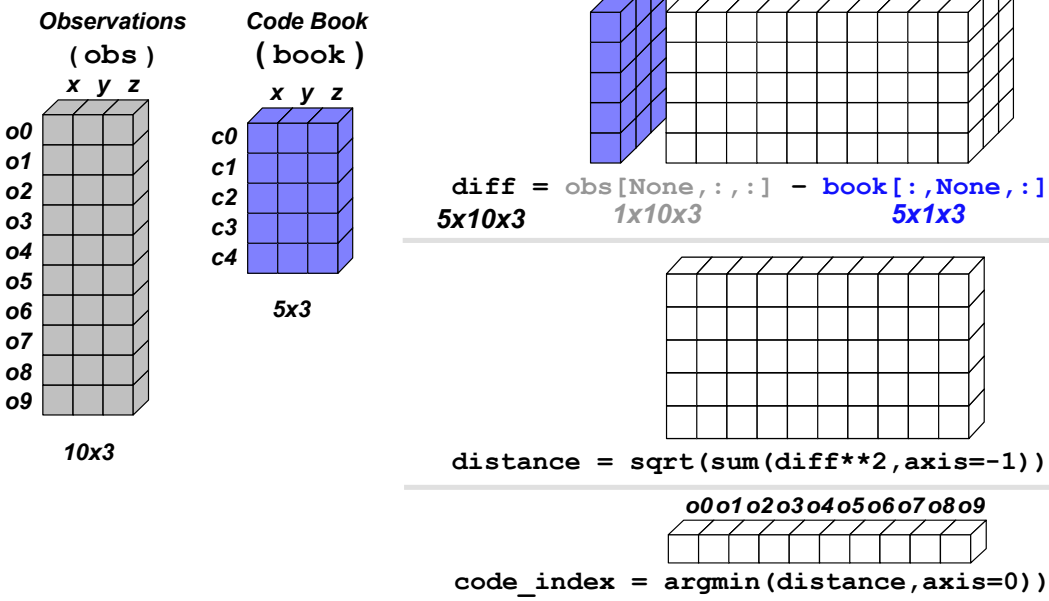





# Vector Quantization Example



# Vector Quantization Example






# 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. 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.

# Pickling

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

```
>>> a = zeros((100,100),dtype=float32)
# total storage
>>> a.nbytes
40000
# standard pickling balloons 4x
>>> ascii = cPickle.dumps(a)
>>> len(ascii)
160061
# binary pickling is very nearly 1x
>>> binary = cPickle.dumps(a,2)
>>> len(binary)
40051
```



# 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 suppress printing small floating point values in scientific notation. The default is False.** |

# 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)
array([  0.000e+00,   1.000e+00,   2.000e+00, ...,
         1.000e+06,   1.000e+06,   1.000e+06])
```

## SUPRESSING 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.])
```



# 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' (or 'print'), 'raise', or 'call'.**

| | |
|---|---|
| **all** | **Set the error handling mode for all error types to the specified value.** |
| **divide** | **Set the error handling mode for 'divide-by-zero' errors.** |
| **over** | **Set the error handling mode for 'overflow' errors.** |
| **under** | **Set the error handling mode for 'underflow' errors.** |
| **invalid** | **Set the error handling mode for 'invalid' floating point errors.** |

# Controlling Error Handling

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

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



# "Composite" Data Structures

```
from numpy import array, dtype, lexsort

particle_dtype = dtype([('mass','f4'), ('velocity', 'f4')])

# This must be a list of tuples.  numpy doesn't like a list of arrays
# or an tuple of tuples.
particles = array([(1,1),
                   (1,2),
                   (2,1),
                   (1,3)],dtype=particle_dtype)

# print the particles
print 'particles (mass, velocity)'
print particles

# lexsort takes keys in reverse order of key specification...
order = lexsort(keys=(particles['velocity'],particles['mass']))

# see demo/mutlitype_array/particle.py
```

# 2D Plotting and Visualization



## Plotting Libraries

- Matplotlib
  http://matplotlib.sf.net
- Chaco
  http://code.enthought.com/chaco
- Dispyl
  http://kim.bio.upenn.edu/~pmagwene/disipyl.html
- Biggles
  http://biggles/sf.net
- Pyngl
  http://www.pyngl.ucar.edu/
- Many more…
  http://www.scipy.org/Topical_Software

# Recommendations

- Matplotlib for day-to-day data exploration.

  Matplotlib has a large community, tons of plot types, and is well integrated into ipython. It is the de-facto standard for 'command line' plotting from ipython.

- Chaco for building interactive plotting applications

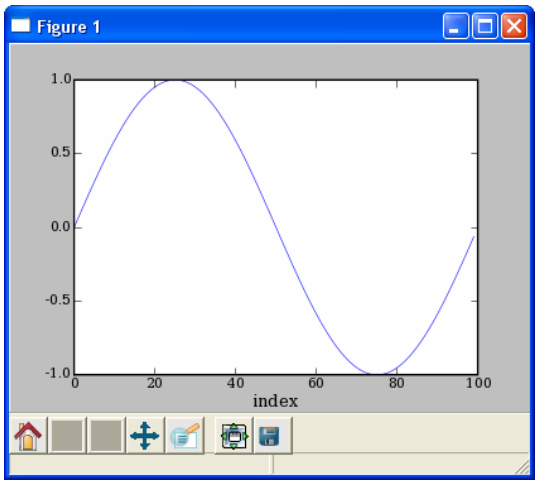
  Chaco is architected for building highly interactive and configurable plots in python. It is more useful as plotting toolkit than for making one-off plots.



# Line Plots

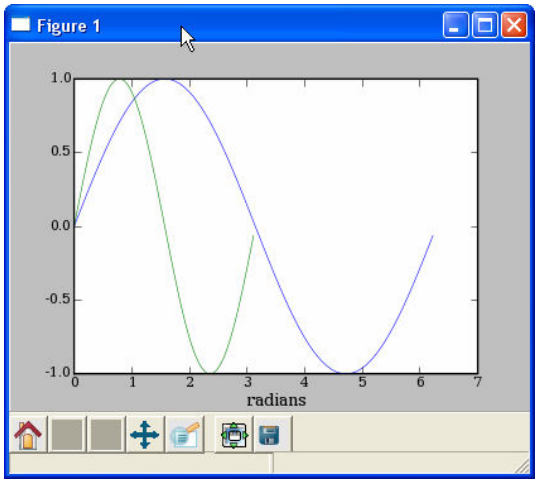
## PLOT AGAINST INDICES

```
>>> x = arange(50)*2*pi/50.
>>> y = sin(x)
>>> plot(y)
>>> xlabel('index')
```




## MULTIPLE DATA SETS

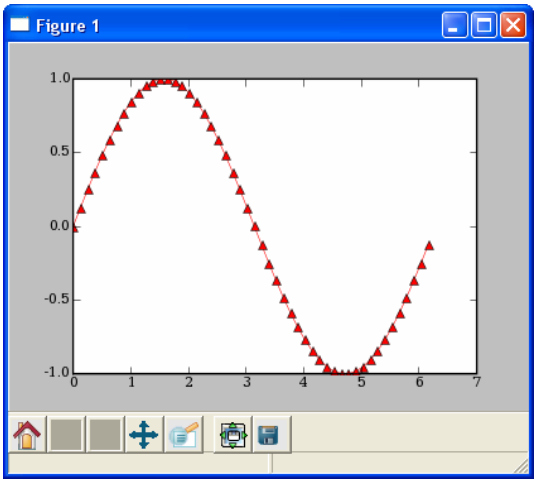
```
>>> plot(x,y,x2,y2)
>>> xlabel('radians')
```



# Line Plots

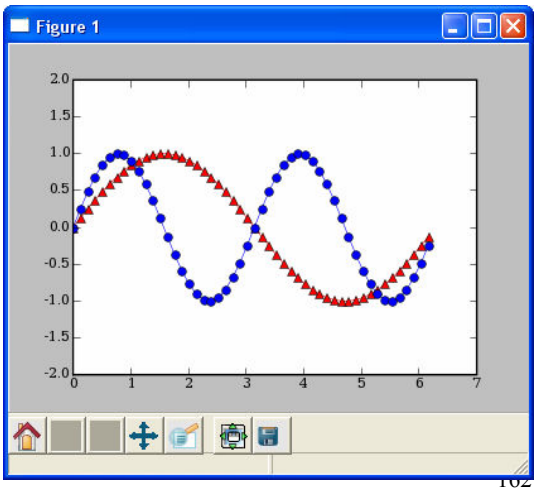
## LINE FORMATTING

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




## MULTIPLE PLOT GROUPS

```
>>> plot(x,y1,'b-o', x,y2), r-^')
>>> axis([0,7,-2,2])
```

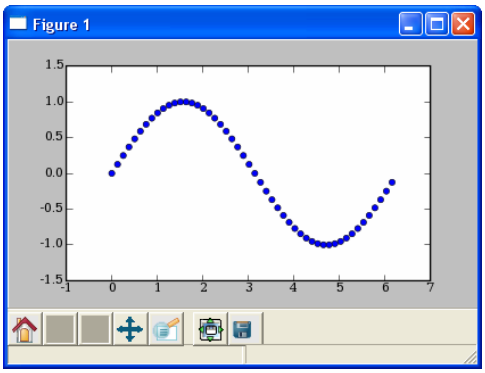





# Scatter Plots

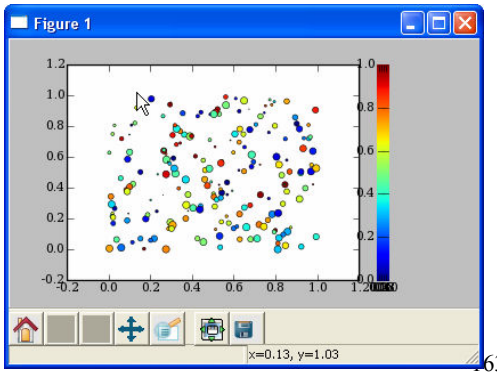
## SIMPLE SCATTER PLOT

```
>>> x = arange(50)*2*pi/50.
>>> y = sin(x)
>>> scatter(x,y)
```




## COLORMAPPED SCATTER

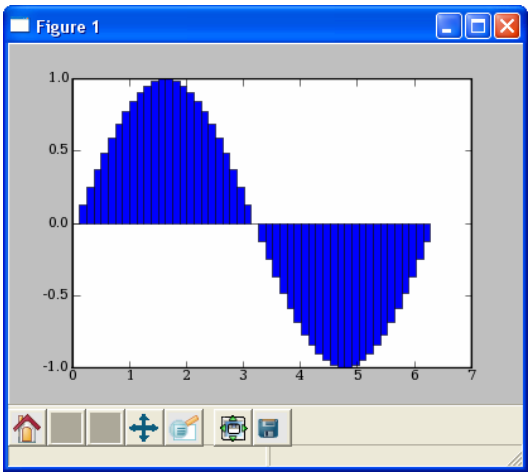
```
# marker size/color set with data
>>> x = rand(200)
>>> y = rand(200)
>>> size = rand(200)*30
>>> color = rand(200)
>>> scatter(x, y, size, color)
>>> colorbar()
```



# Bar Plots

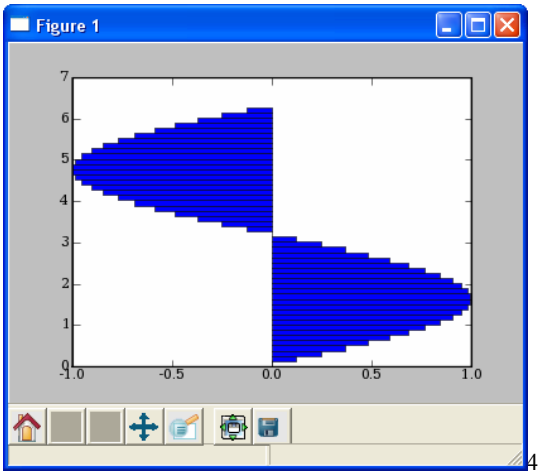
## BAR PLOT

```
>>> bar(x,sin(x),
...      width=x[1]-x[0])
```




## HORIZONTAL BAR PLOT

```
>>> bar(x,sin(x),
...      height=x[1]-x[0])
```





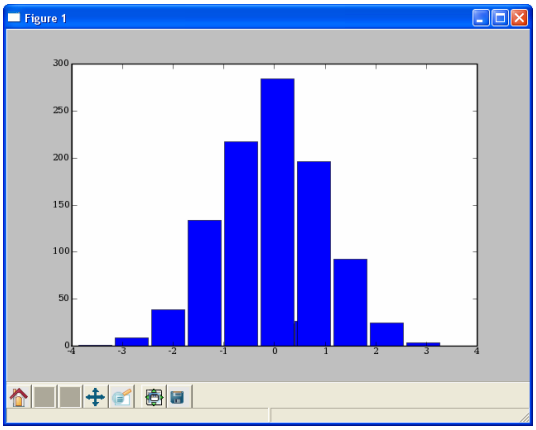

# Bar Plots

## DEMO/MATPLOTLIB_PLOTTING/BARCHART_DEMO.PY

# HISTOGRAMS

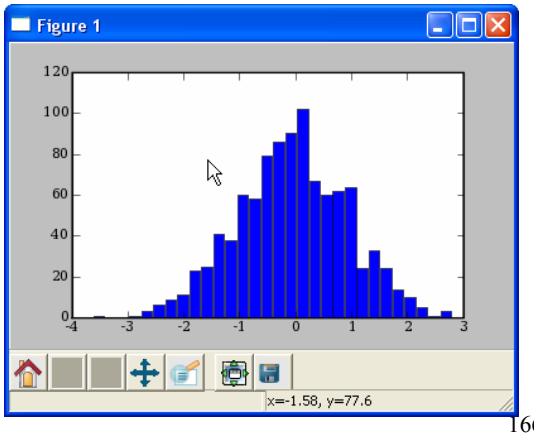
## HISTOGRAM

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




## HISTOGRAM 2

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

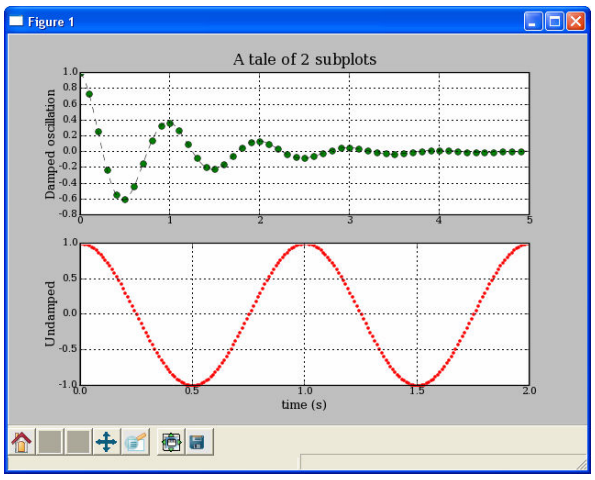





# Multiple Plots using Subplot

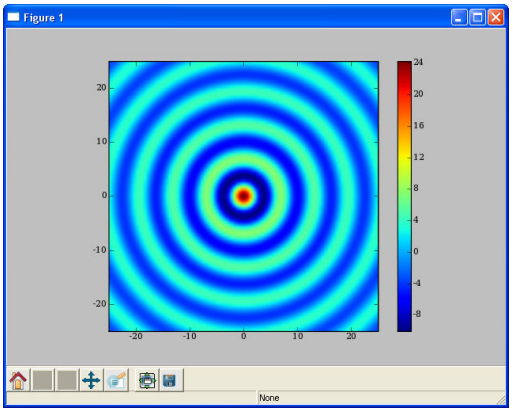
## DEMO/MATPLOTLIB_PLOTTING/EXAMPLES/SUBPLOT_DEMO.PY

```
def f(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
    return multiply(s1,e1)

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)
t3 = arange(0.0, 2.0, 0.01)

subplot(211)
l = plot(t1, f(t1), 'bo', t2, f(t2),
         'k--')
setp(l, 'markerfacecolor', 'g')
grid(True)
title('A tale of 2 subplots')
ylabel('Damped oscillation')

subplot(212)
plot(t3, cos(2*pi*t3), 'r.')
grid(True)
xlabel('time (s)')
ylabel('Undamped')
show()
```



# Image Display

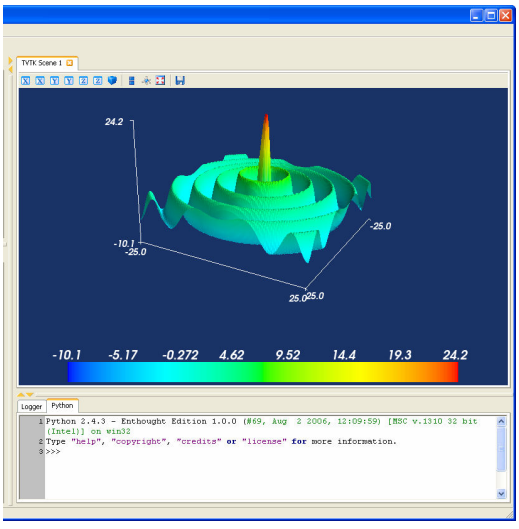
```
# Create 2d array where values
# are radial distance from
# the center of array.
>>> from numpy import mgrid
>>> from scipy import special
>>> x,y = mgrid[-25:25:100j,
...             -25:25:100j]
>>> r = sqrt(x**2+y**2)
# Calculate bessel function of
# each point in array and scale
>>> s = special.j0(r)*25

# Display surface plot.
>>> imshow(s, extent=[-25,25,-25,25])
>>> colorbar()
```






# 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 enthought.mayavi.tools \
    import mlab
>>> mlab.surf(x,y,s)
>>> mlab.scalarbar()
>>> mlab.axes()
```



ENTHOUGHT

# SciPy



ENTHOUGHT

# Overview

- Available at www.scipy.org
- Open Source BSD Style License
- 34 svn “committers” to the project

**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)
- Linear Algebra (scipy.linalg)
- Input/Output (scipy.io)
- Statistics (scipy.stats)
- Distributed Computing (scipy.cow)
- Fast Execution (weave)
- Clustering Algorithms (scipy.cluster)
- Sparse Matrices (scipy.sparse)

# Input and Output

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

Before you use capabilities of scipy.io be sure that pickle, pytables, 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

For IO example,
see speed_of_light.py demo.s



# 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 Numpy 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
```

# 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

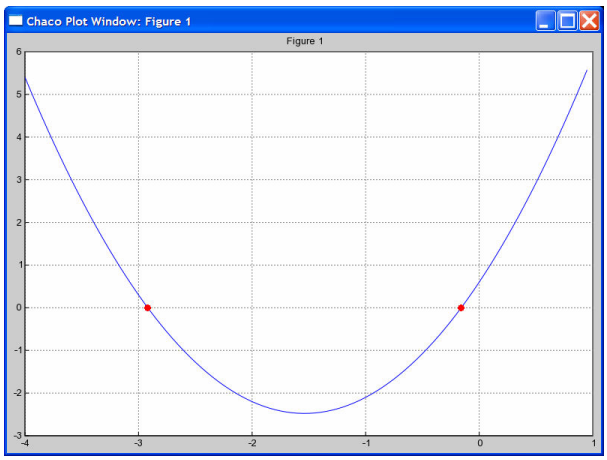
```
>>> p = poly1d([1,-2,4])
>>> print p
 2
x - 2 x + 4

>>> 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

## FINDING THE ROOTS OF A POLYNOMIAL

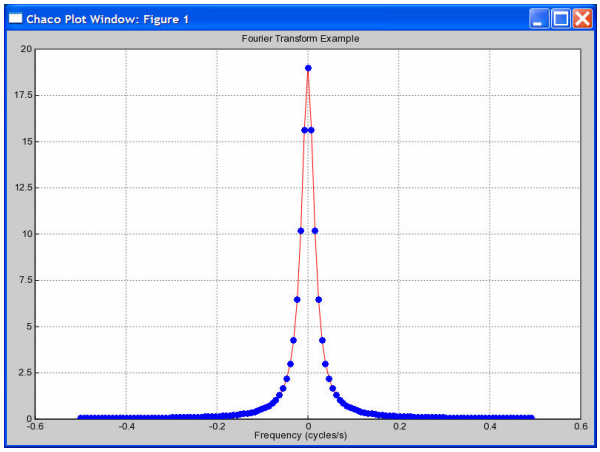
```
>>> p = poly1d([1.3,4,.6])
>>> print p
     2
1.3 x + 4 x + 0.6
>>> x = r_[-4:1:0.05]
>>> y = p(x)
>>> plot(x,y,'-')
>>> hold(True)
>>> r = p.roots
>>> s = p(r)
>>> r
array([-0.15812627, -2.9187968 ])
>>> plot(r.real,s.real,'ro')
```



# FFT

## 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)
>>> f = fftshift(f)
>>> plot(f, fftshift(X.real),'r-',
...      f, fftshift(Xexact.real),'bo')
>>> title('Fourier Transform Example')
>>> xlabel('Frequency (cycles/s)')
>>> axis(-0.6,0.6, 0, 20)
```






# 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)

# Linear Algebra

## LU FACTORIZATION

```
>>> from scipy import linalg
>>> a = array([[1,3,5],
...            [2,5,1],
...            [2,3,6]])
# time consuming factorization
>>> lu, piv = linalg.lu_factor(a)

# fast solve for 1 or more
# right hand sides.
>>> b = array([10,8,3])
>>> linalg.lu_solve((lu, piv), b)
array([-7.82608696,  4.56521739,
        0.82608696])
```

## EIGEN VALUES AND VECTORS

```
>>> from scipy import linalg
>>> a = array([[1,3,5],
...            [2,5,1],
...            [2,3,6]])
# compute eigen values/vectors
>>> vals, vecs = linalg.eig(a)
# print eigen values
>>> vals
array([ 9.39895873+0.j,
       -0.73379338+0.j,
        3.33483465+0.j])
# eigen vectors are in columns
# print first eigen vector
>>> vecs[:,0]
array([-0.57028326,
       -0.41979215,
       -0.70608183])
# norm of vector should be 1.0
>>> linalg.norm(vecs[:,0])
1.0
```



# Matrix Objects

## STRING CONSTRUCTION

```
>>> from numpy import mat
>>> a = mat('[1,3,5;2,5,1;2,3,6]')
>>> a
matrix([[1, 3, 5],
        [2, 5, 1],
        [2, 3, 6]])
```

## TRANSPOSE ATTRIBUTE

```
>>> a.T
matrix([[1, 2, 2],
        [3, 5, 3],
        [5, 1, 6]])
```

## INVERTED ATTRIBUTE

```
>>> a.I
matrix([[-1.1739,  0.1304,  0.956],
        [ 0.4347,  0.1739, -0.391],
        [ 0.1739, -0.130,  0.0434]
       ])
# note: reformatted to fit slide
```

## DIAGONAL

```
>>> a.diagonal()
matrix([[1, 5, 6]])
>>> a.diagonal(-1)
matrix([[3, 1]])
```

## SOLVE

```
>>> b = mat('10;8;3')
>>> a.I*b
matrix([[-7.82608696],
        [ 4.56521739],
        [ 0.82608696]])

>>> from scipy import linalg
>>> linalg.solve(a,b)
matrix([[-7.82608696],
        [ 4.56521739],
        [ 0.82608696]])
```

# Special Functions

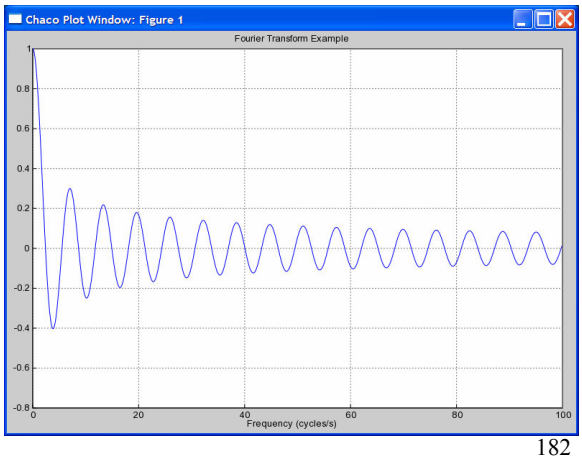
## scipy.special

**Includes over 200 functions:**

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

### FIRST ORDER BESSEL EXAMPLE

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



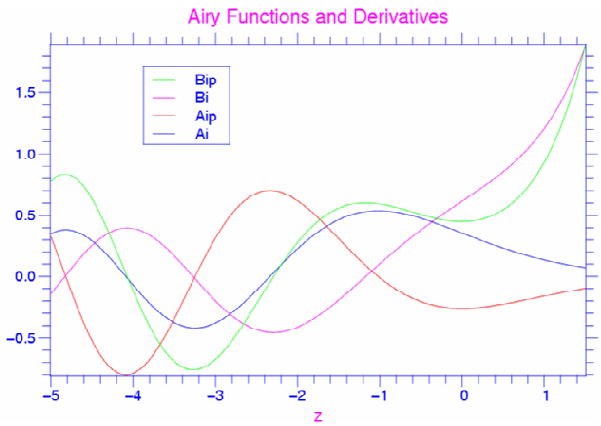



# Special Functions

## 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

## scipy.special --- Vectorizing a function

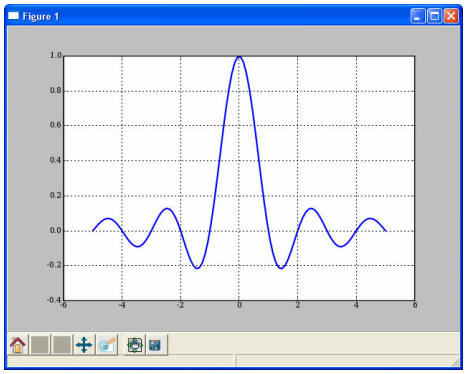
**Example**

```
# 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
```

## SOLUTION

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

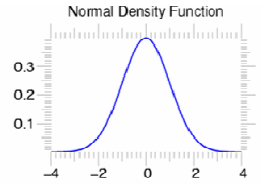
```
# attempt
>>> sinc([1.3,1.5])
TypeError: can't multiply
sequence to non-int
>>> x = r_[-5:5:100j]
>>> y = vsinc(x)
>>> plot(x, y)
```

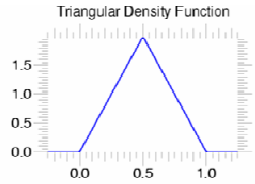


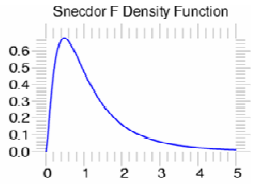
# Statistics

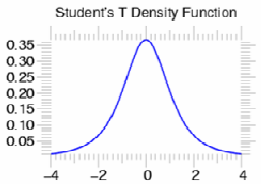
## scipy.stats --- CONTINUOUS DISTRIBUTIONS

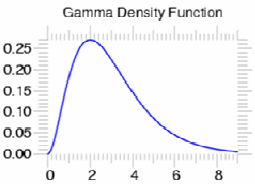
over 80 continuous distributions!

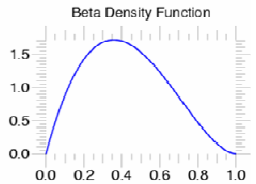


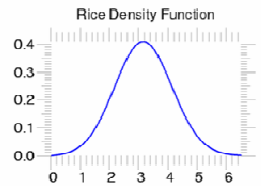



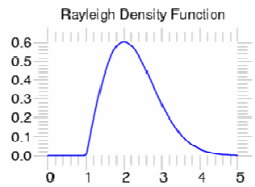



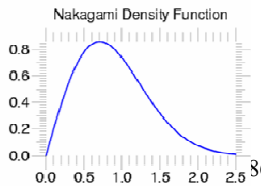




**METHODS**

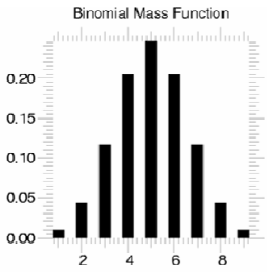
- `pdf`
- `cdf`
- `rvs`
- `ppf`
- `stats`

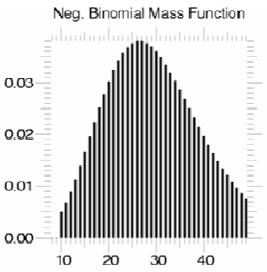


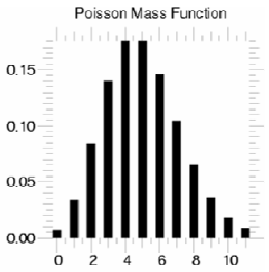
# Statistics

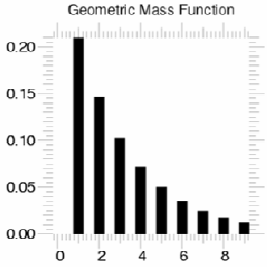
## scipy.stats --- Discrete Distributions

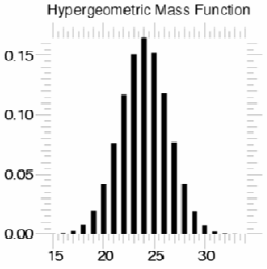
10 standard discrete distributions (plus any arbitrary finite RV)

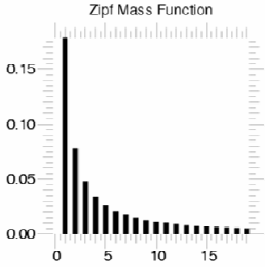


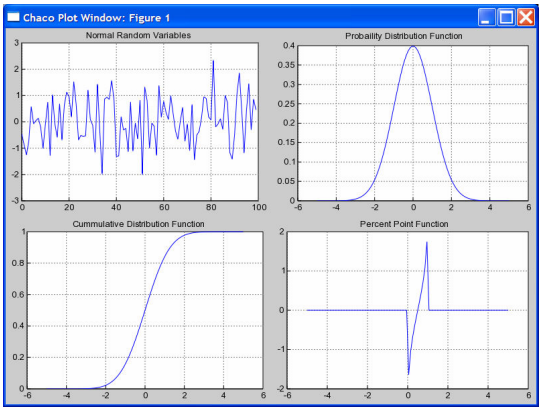

**METHODS**

- `pdf`
- `cdf`
- `rvs`
- `ppf`
- `stats`

# Using stats objects

## DISTRIBUTIONS

```
# Sample normal dist. 100 times.
>>> samp = stats.norm.rvs(size=100)

>>> x = r_[-5:5:100j]
# Calculate probability dist.
>>> pdf = stats.norm.pdf(x)
# Calculate cummulative Dist.
>>> cdf = stats.norm.cdf(x)
# Calculate Percent Point Function
>>> ppf = stats.norm.ppf(x)
```






# 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

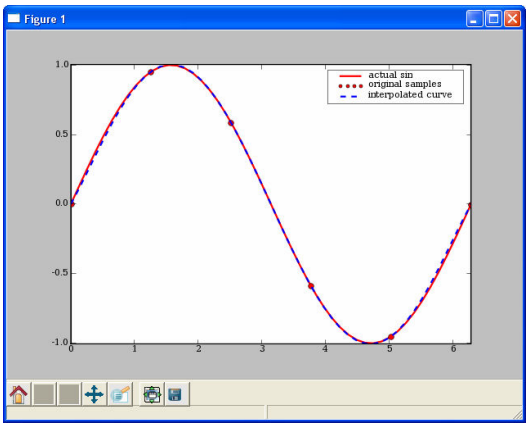
**scipy.interpolate --- General purpose Interpolation**

- **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



# 1D Spline Interpolation

```
# demo/interpolate/spline.py
from scipy.interpolate import UnivariateSpline
from matplotlib.pylab import plot, axis,legend

# sample values
x = linspace(0,2*pi,6)
y = sin(x)

# Create a spline class for interpolation.
# k=5 sets to 5th degree spline.
spline_fit = UnivariateSpline(x,y,k=5)
xx = linspace(0,2*pi, 50)
yy = spline_fit(xx)
```




```
# display the results.
plot(xx, sin(xx), 'r-', x,y,'ro',xx,yy, 'b--',linewidth=2)
axis('tight')
legend(['actual sin', 'original samples', 'interpolated curve'])
```

# 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**

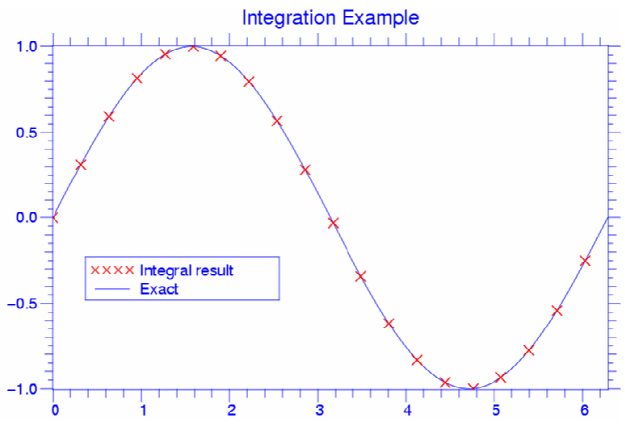
  `integrate.quad` (general purpose), `integrate.dblquad` (double integration), `integrate.tplquad` (triple integration), `integrate.fixed_quad` (fixed order Gaussian integration), `integrate.quadrature` (Gaussian quadrature to tolerance), `integrate.romberg` (Romberg)



# Integration

## scipy.integrate --- Example

```
# Compare sin to integral(cos)
>>> 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)
>>> plot(x,y,x2,y2,'rx')
>>> legend(['Exact',
...          'Integral Result'])
```



# Signal Processing

## scipy.signal --- Signal and Image Processing

### What's Available?

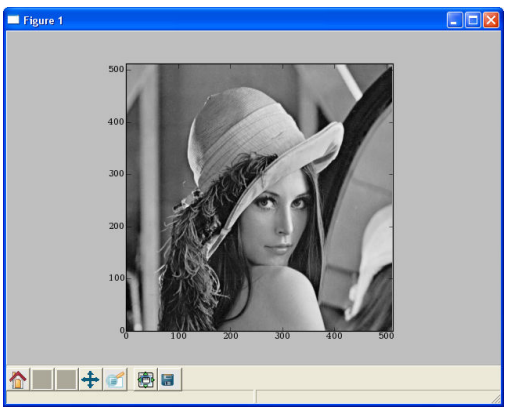
- **Filtering**
  - General 2-D Convolution (more boundary conditions)
  - N-D convolution
  - B-spline filtering
  - N-D Order filter, N-D median filter, faster 2d version,
  - IIR and FIR filtering and filter design
- LTI systems
  - System simulation
  - Impulse and step responses
  - Partial fraction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



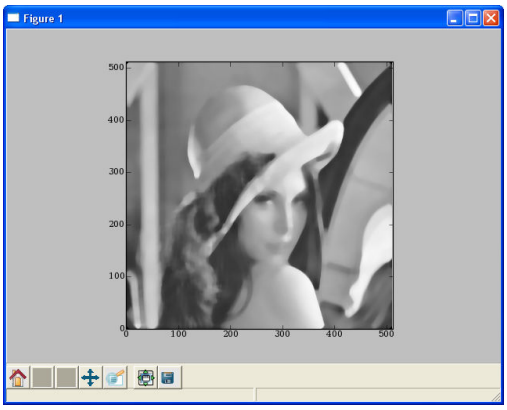
# Image Processing

```
# The famous lena image is packaged with scipy
>>> from scipy import lena, signal
>>> lena = lena().astype(float32)
>>> imshow(lena, cmap=cm.gray)
# Blurring using a median filter
>>> fl = signal.medfilt2d(lena, [15,15])
>>> imshow(fl, cmap=cm.gray)
```

**LENA IMAGE**



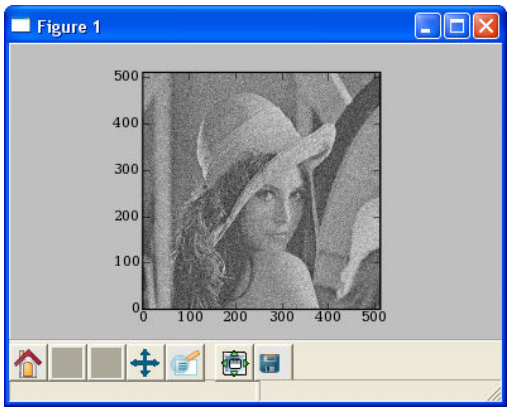

**MEDIAN FILTERED IMAGE**



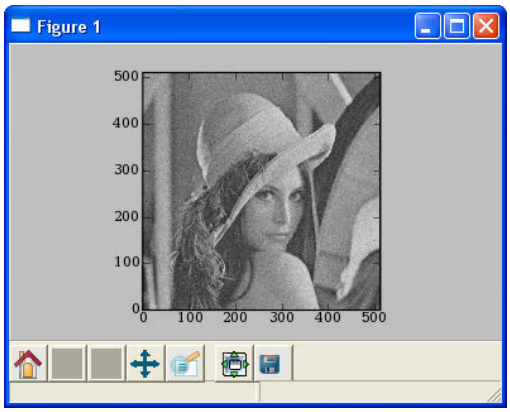
# Image Processing

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

**NOISY IMAGE**




**FILTERED IMAGE**

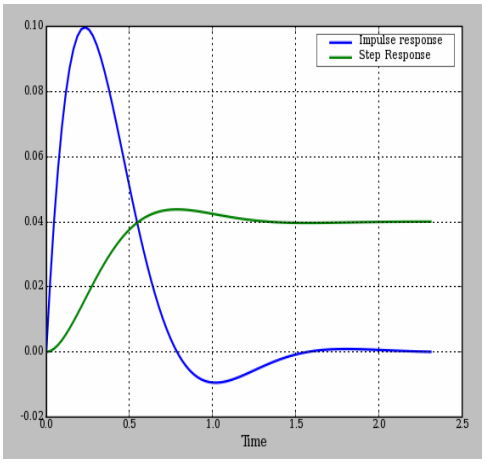





# LTI Systems

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

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



# 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)

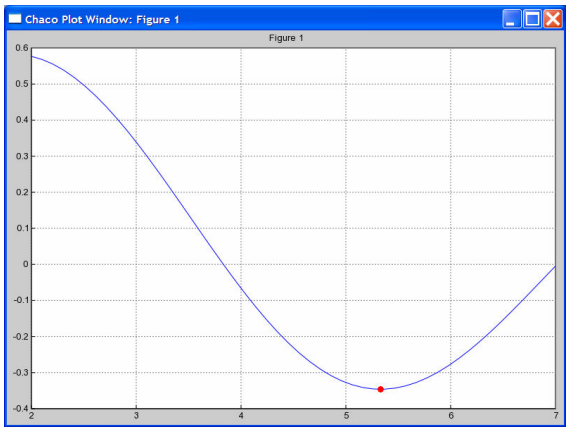


# Optimization

ENTHOUGHT

## 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)
>>> plot(x,j1x,'-')
>>> hold(True)
>>> x_min = fminbound(j1,4,7)
>>> j1_min = j1(x_min)
>>> plot([x_min],[j1_min],'ro')
```



# Optimization

## EXAMPLE: SOLVING NONLINEAR EQUATIONS

Solve the non-linear equations

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

```
>>> 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]
```

starting location for search



# 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`

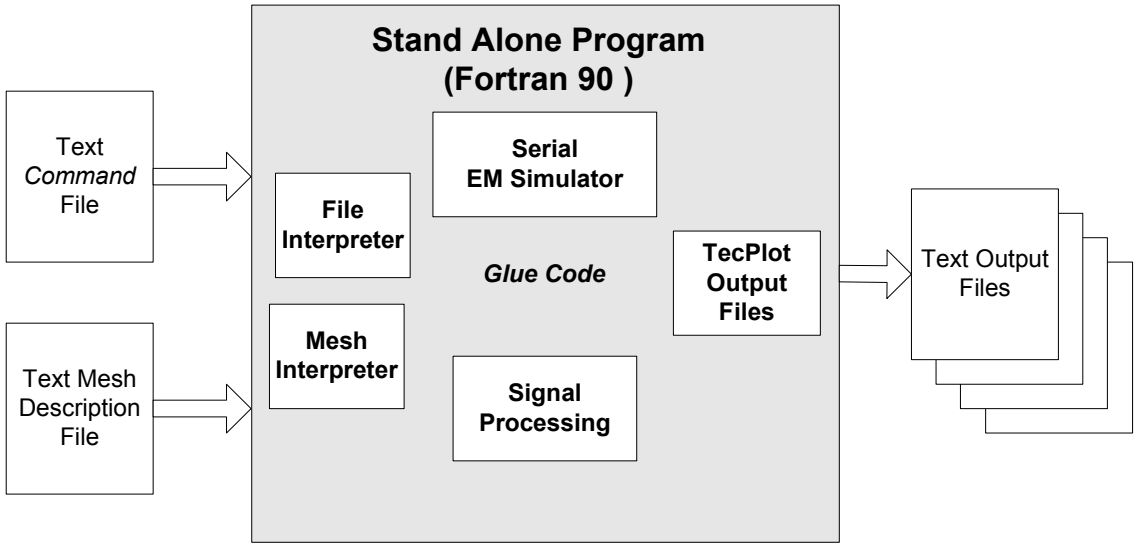


# Python as “Glue”

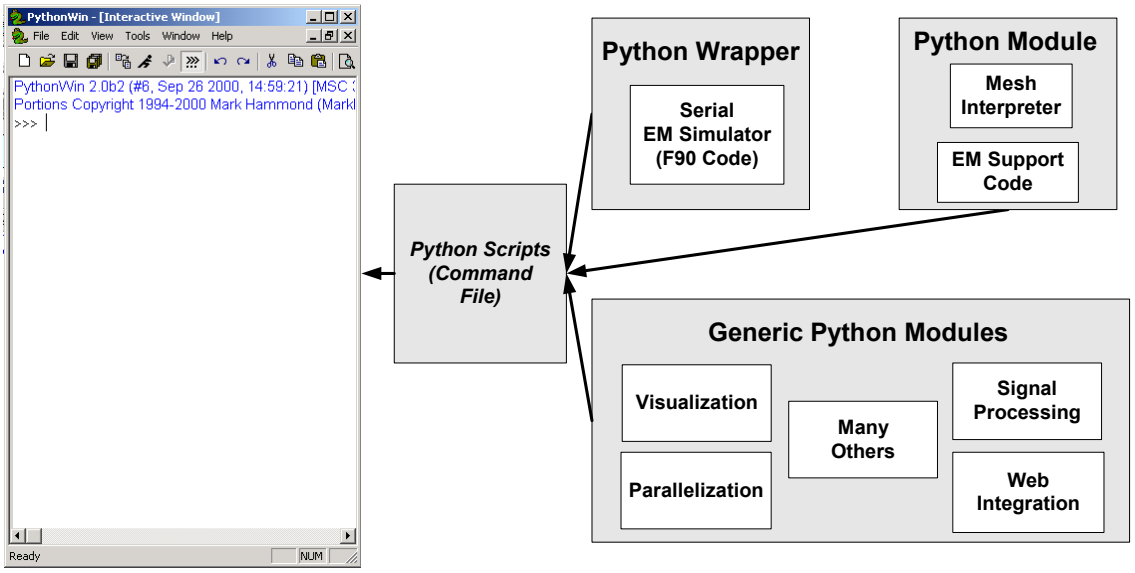
# A Case Study for Transitioning from F90 to Python



## Legacy System



# The Python Approach






# Simple Input Script

```
import em
from em.material import air,yuma_soil_5_percent_water

# 1. Load standard mesh file
mesh = em.standard_mesh.small_sphere()

# 2. Create layered halfspace from predefined materials.
layers = [air,yuma_soil_5_percent_water]
environment = em.layered_media(layers)

# 3. Build MoM solver based on the mesh and environement.
solver = em.mom(environment,mesh)

# 4. Solve and print results at multiple frequencies.
freqs = arange(100e6,201e6,10e6)
for f in freqs:
    source = em.plane_wave(theta=0,phi=0,freq=f)
    solver.solve_currents(source)
    this_rcs = solver.calc_rcs( [[0.,0.]] )
    print f, this_rcs[0]
```

# Electromagnetics Example

(1) Parallel simulation
(2) Create plot
(3) Build HTML page
(4) FTP page to Web Server
(5) E-mail users that results are available.

```
def ex8():
    """ 1. Parallel solve for rcs at multiple freqs.
        2. Plot rcs.
        3. Make into HTML page.
        4. Upload to web server using FTP.
        5. Mail notification that the results are done. """
    import em, time,plt, scipy.cow
    from em.material import air,yuma_soil_5_percent_water
    # 1. Solve for the RCS
    print 'Solving for RCS in parallel'
    #    Load Mesh
    mesh = em.standard_mesh.small_sphere()
    mesh.offset((0,0,-0.15))
    #    Create a machine cluster for parallel processing
    ml = [('10.0.2.1',10000),('10.0.2.2',10000),
          ('10.0.2.3',10000),('10.0.2.4',10000)]
    cluster = scipy.cow.machine_cluster(ml)
    # create the layered halfspace
    layers = [air, yuma_soil_5_percent_water]
    environment = em.layered_media(layers)
    # create incident plane wave
    t,p = array[(30.,30.)]* pi/180.
    source = em.plane_wave(theta=t,phi=p,freq=300e6)
    freqs = arange(100e6,996e6,28e6)
    #    Solve in parallel
    tl = time.time()
    solver = em.mom(environment,mesh)
    back_scatter = em.monostatic.parallel_old(solver,freqs,
                                              look angles,
                                              cluster=cluster)
    parallel_time = time.time() - tl
    parallel_time_per_freq = parallel_time/len(freqs)
    #    Extract vertical polarization from results
    vv = array(back scatter)[:,0,0]
    #    Comparison Serial Run
    tl = time.time();
    solver.solve_currents(source)
    serial time = time.time() - tl
    #2. Plot rcs
```

```
plt.plot(freqs/1e6,vv)
t= 'Monostatic VV RCS for .2m buried sphere: theta=30, phi=30'
plt.title(t)
plt.xtitle('frequency (MHz)')
plt.ytitle('RCS (dB)')
plt.save('rcs.png')
#3. Build Simple HTML file with it.
html = """<hl> Simulation Output </hl>
        <img src="rcs.png">  """
#4. FTP it to our server
server = 'n0'
import ftplib, cStringIO
img = open('rcs.png','rb')
html_file = cStringIO.StringIO(html)
ftp = ftplib.FTP(server,user='ej',passwd='xxx')
ftp.cwd('public_html') #go to web directory
try:
    ftp.sendcmd('DELE rcs.png')
    ftp.sendcmd('DELE rcs.html')
except:
    pass
ftp.storbinary('STOR rcs.png', img,1024)
ftp.storlines('STOR rcs.html', html_file)
img.close()
ftp.quit()
#5. Mail me a notification that the run is finished.
import smtplib
msg = """Hello Eric,
         Your rcs simulation is done.
            Serial time per frequency:     %3.3f sec
            Parallel time per frequency:   %3.3f sec
            factor of speed up:            %3.3f
         You may view the RCS vs. Freq. graph at:
            http://n0/~ej/rcs.html
      """ % (serial_time,parallel_time_per_freq
             serial_time/parallel_time_per_freq)
mailer = smtplib.SMTP(server)
mailer.sendmail('em simulator@'+server,['ej@'+server],msg)
```



# General issues with Fortran

- Fortran 77 encouraged the use of a large number of globals.
  - causes *states* within wrapped modules.
  - makes threading difficult
- Legacy code is often not very modular.
  - occasionally one or two very long functions that do absolutely everything.
- Fortran 77 doesn't handle memory allocation well
- Fortran 90 objects don't map easily to C structures (the underpinnings of Python)

# Global variables – here's the rub

**MODULE WITH GLOBALS CALLED SAFELY…**

```
# Global variables create states in modules
>>> import f90_module
# f1() returns an array and also quietly sets a global variable zzz
>>> a = f90_module.f1(5,6) # → zzz = 5
# f2() uses a AS WELL AS zzz
>>> f90_module.f2(a)
xxx # returns some value
```

**AND THE HAVOC AN INTERMEDIATE CALL CAN CAUSE**

```
# Users of interactive interpreters and scripting languages can and
# do call functions in a variety of orders.  As a result, they can
# unknowingly corrupt the internal state of the module.
>>> a = f90_module.f1(5,6)     # → zzz = 5
# user makes additional call to f1 with different variables
>>> b = f90_module.f1(20,30)   # → zzz = 20
# Now user calls f2 expecting to get same answer as above
>>> f90_module.f2(a)
yyy #Yikes! result is different because globals were different
```



# Solutions to global variables

1. Remove the ability of calling functions out of order by wrapping functions at the highest level.

```
# users call the following instead of f1 and f2 individually
def wrapper_for_f1_and_f2(a,b):
    c = f90_module.f1_QUITELY_SETS_LOCAL_VARIABLES(a,b)
    return f90_module.f2_that_uses_a_AND_GLOBAL_VARIABLES(c)
```

2. Get rid of global variables and include them in the argument list for functions. (The best option, but potentially a lot of work)

```
# Return the affected global variables b, c, and d.
>>> a,b,c,d = f90_module.f1(5,6)
# Now call f2 passing in the variables that used to be global
>>> f90_module.f2 (a,b,c,d)
```

# Problems with non-modularity

Generally only want to wrap Fortran simulation engine and leave pre and post processing to Python.

If input/engine/output are all closely coupled or live in one huge function (worst case), it is extremely difficult to find convenient places in the code to *hook* into the simulator



# Solutions to modularity

For codes not written with modularity in mind, this is a lot of work. Try and find logical boundaries in the code and rewrite long functions that do several things as multiple functions that do only one thing.

Dividing the code into smaller functions has other advantages also. It generally leads to more readable and robust code, and it facilitates unit testing. A suite of unit tests can be run on each of the functions after code changes to quickly locate bugs.
See PyUnit at http://pyunit.sourceforge.net/

# The starting place for our code

Fortunately, Norbert and Larry had written a very good piece of F90 software that is fairly modular and had only a few global variables – the impedance matrix being the main one.

We chose to expose only three functions:

**`tpm_hs_cfie_zcomplete()`**

Given a problem description, this function creates the impedance matrix.

**`tpm_hs_pwi_cfie_vdrive()`**

Given a problem description and a source, calculate the right hand side.

**`tpm_hs_ffs_pec_smatrix()`**

Given the problem description and currents, calculate the far field in a given direction.



# Memory allocation & generic math

- All memory allocation is handled in Python
- Mathematically general operations are left to Python.
  - Linear Algebra is handled by a wrapper to netlib's LAPACK.
  - FFT's are handled by a wrapper to netlib's FFTPACK. FFTW wrappers are also available.

# Long argument lists

Fortran encourages long argument lists – unless you use globals which we did not allow. One of our typical functions has *21* arguments. This is unmanageable for an interactive interface.

```
subroutine tpm_hs_pwi_cfie_vdrive (alpha,i12,e0,cfreq,thi,phi,
                                   epstj,muej,halfspace,tnj,
                                   tnjmax,tnn,tnnmax,tnp,
                                   tnpmax, tx3j,tj3p,tn6p,vmth,
                                   vmph,errlog,errtxt)
```

Black arguments affect the CFIE equation.

Red arguments describe the source.

Green arguments describe the environment.

Orange arguments describe the target mesh.

Blue arguments are return values from the function



# Solution to long argument lists

- Python wrappers alone remove the need for 9 arguments by removing array indices and return arguments.

```
vmth,vmph,errlog,errtxt = tpm_hs_pwi_cfie_vdrive(alpha,i12,
                                  e0,cfreq,thi,phi,epstj,muej,
                                  halfspace,tnn,tx3j,tj3p,tn6p)
```

- Using classes to group the other arguments can further simply the interface and facilitate user interaction.

# Object oriented view of problem

- Problem Definition
  - Environment
    - free-space/ half-space, material types
  - Target description
    - mesh and perhaps material types
- Source Signal
    - near/far, time/frequency domain
- An algorithm for solving the problem
    - MoM, MLFMA, FDTD, etc

The words highlighted in **blue** indicate objects that are used as objects in the Python interface to the Duke MoM code.



# Using the OO interface

```
# 1. Specify the mesh you would like to use.
>>> import em
>>> mesh = em.standard_mesh.small_sphere()
# 2. Perhaps burry it .15 meters underground.
#    (sphere radius is 0.10 meters)
>>> mesh.offset((0,0,-0.15))
# 3. Specify the layers of materials in the environment.
>>> from em.material import air,yuma_soil_5_percent_water
>>> layers = [air, yuma_soil_5_percent_water]
>>> environment = em.layered_media(layers)
# 4. Build a MoM solver for this mesh and environment.
>>> solver = em.mom(environment,mesh)
# 5. Specify a source frequency and angle (in radians)
#    and solve for the currents on the mesh.
>>> source = em.plane_wave(theta=0,phi=0,freq=50e6)
>>> solver.solve_currents(source)
# 6. Post process.  Here we'll calculate the monostatic
# scattered field.
>>> angles = [[0.,0.]]
>>> rcs = solver.calc_rcs(angles)
```

# Review

- Reduce usage of globals as much as possible when ever practical.
- Divide large functions in to smaller pieces to provide more control over simulator.
- Use Python to allocate memory arrays. f2py will handle converting them to Fortran arrays.
- Be careful about using F90 constructs. They are difficult to wrap.
- Object Oriented interfaces hide low level array representations and allow users to work with the actual components of the problem.



# Tools

- C/C++ Integration
  - SWIG www.swig.org
  - ctypes python standard library
  - Pyrex nz.cosc.canterbury.ac.nz/~greg/python/Pyrex
  - boost www.boost.org/libs/python/doc/index.html
  - weave www.scipy.org/site_content/weave
- FORTRAN Integration
  - f2py cens.ioc.ee/projects/f2py2e/ (now part of numpy)
  - PyFort pyfortran.sourceforge.net

# Weave



## weave

- `weave.blitz()`

  Translation of Numeric array expressions to C/C++ for fast execution

- `weave.inline()`

  Include C/C++ code directly in Python code for on-the-fly execution

- `weave.ext_tools`

  Classes for building C/C++ extension modules in Python

# weave.inline

```
>>> from scipy import weave
>>> a=1
>>> weave.inline('std::cout << a << std::endl;',['a'])
sc_f08dc0f70451ecf9a9c9d4d0636de3670.cpp
   Creating library <snip>
1
>>> weave.inline('std::cout << a << std::endl;',['a'])
1
>>> a='qwerty'
>>> weave.inline('std::cout << a << std::endl;',['a'])
sc_f08dc0f70451ecf9a9c9d4d0636de3671.cpp
   Creating library <snip>
qwerty
>>> weave.inline('std::cout << a << std::endl;',['a'])
qwerty
```



# Support code example

```
>>> from scipy import weave
>>> a = 1
>>> support_code = 'int bob(int val) { return val;}'
>>> weave.inline('return_val = bob(a);',['a'],support_code=support_code)
sc_19f0a1876e0022290e9104c0cce4f00c0.cpp
   Creating library <snip>
1
>>> a = 'string'
>>> weave.inline('return_val = bob(a);',['a'],support_code = support_code)
sc_19f0a1876e0022290e9104c0cce4f00c1.cpp
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\sc_19f0a1876e0022290e9104c0cce4
f00c1.cpp(417) : error C2664: 'bob' : cannot convert parameter 1 from 'class Py:
:String' to 'int' No user-defined-conversion operator available that can
perform this conversion, or the operator cannot be called
Traceback (most recent call last):
    <snip>
weave.build_tools.CompileError: error: command '"C:\Program Files\Microsoft Visu
al Studio\VC98\BIN\cl.exe"' failed with exit status 2
```

# ext_tools example

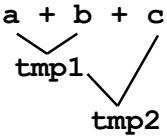
```
import string
from weave import ext_tools
def build_ex1():
    ext = ext_tools.ext_module('_ex1')
    # Type declarations- define a sequence and a function
    seq = []
    func = string.upper
    code = """
           py::tuple args(1);
           py::list result(seq.length());
           for(int i = 0; i < seq.length();i++)
           {
              args[0] = seq[i];
              result[i] = func.call(args);
           }
           return_val = result;
           """
    func = ext_tools.ext_function('my_map',code,['func','seq'])
    ext.add_function(func)
    ext.compile()

try:
    from _ex1 import *
except ImportError:
    build_ex1()
    from _ex1 import *

if __name__ == '__main__':
    print my_map(string.lower,['asdf','ADFS','ADSD'])
```



# Efficiency Issues

## PSEUDO C FOR STANDARD NUMERIC EVALUATION

```
>>> c = a + b + c
```




```
// c code
// tmp1 = a + b
tmp1 = malloc(len_a * el_sz);
for(i=0; i < len_a; i++)
    tmp1[i] = a[i] + b[i];
// tmp2 = tmp1 + c
tmp2 = malloc(len_c * el_sz);
for(i=0; i < len_c; i++)
    tmp2[i] = tmp1[i] + c[i];
```

## FAST, IDIOMATIC C CODE

```
>>> c = a + b + c
```

```
// c code
// 1. loops "fused"
// 2. no memory allocation
for(i=0; i < len_a; i++)
    c[i] = a[i] + b[i] + c[i];
```

# Finite Difference Equation

## MAXWELL'S EQUATIONS: FINITE DIFFERENCE TIME DOMAIN (FDTD), UPDATE OF X COMPONENT OF ELECTRIC FIELD

$$E_x = \frac{1 - \frac{\sigma_x \Delta t}{2\varepsilon_x}}{1 + \frac{\sigma_x \Delta t}{2\varepsilon_x}} E_x + \frac{\Delta t}{\varepsilon_x + \frac{\sigma_x \Delta t}{2}} \frac{dH_z}{dy} - \frac{\Delta t}{\varepsilon_x + \frac{\sigma_x \Delta t}{2}} \frac{dH_y}{dz}$$

## PYTHON VERSION OF SAME EQUATION, PRE-CALCULATED CONSTANTS

```
ex[:,1:,1:] =   ca_x[:,1:,1:]   * ex[:,1:,1:]
              + cb_y_x[:,1:,1:] * (hz[:,1:,1:] - hz[:,:-1,:])
              - cb_z_x[:,1:,1:] * (hy[:,1:,1:] - hy[:,1:,:-1])
```



# weave.blitz

weave.blitz compiles array expressions to C/C++ code using the Blitz++ library.

## WEAVE.BLITZ VERSION OF SAME EQUATION

```
>>> from scipy import weave
>>> # <instantiate all array variables...>
>>> expr = "ex[:,1:,1:] =   ca_x[:,1:,1:]   * ex[:,1:,1:]"\
           "+ cb_y_x[:,1:,1:] * (hz[:,1:,1:] - hz[:,:-1,:])"\
           "- cb_z_x[:,1:,1:] * (hy[:,1:,1:] - hy[:,1:,:-1])"

>>> weave.blitz(expr)
 < 1. translate expression to blitz++ expression>
 < 2. compile with gcc using array variables in local scope>
 < 3. load compiled module and execute code>
```

# weave.blitz benchmarks

| Equation | Numeric (sec) | Inplace (sec) | compiler (sec) | Speed Up |
|---|---|---|---|---|
| Float (4 bytes) | | | | |
| a = b + c (512,512) | 0.027 | 0.019 | 0.024 | 1.13 |
| a = b + c + d (512x512) | 0.060 | 0.037 | 0.029 | 2.06 |
| 5 pt. avg filter (512x512) | 0.161 | - | 0.060 | 2.68 |
| FDTD (100x100x100) | 0.890 | - | 0.323 | 2.75 |
| Double (8 bytes) | | | | |
| a = b + c (512,512) | 0.128 | 0.106 | 0.042 | 3.05 |
| a = b + c + d (512x512) | 0.248 | 0.210 | 0.054 | 4.59 |
| 5 pt. avg filter (512x512) | 0.631 | - | 0.070 | 9.01 |
| FDTD (100x100x100) | 3.399 | - | 0.395 | 8.61 |

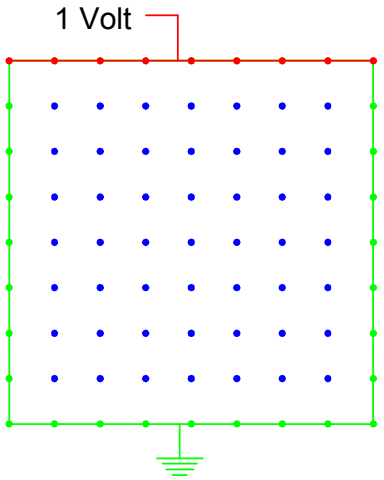
Pentium II, 300 MHz, Python 2.0, Numeric 17.2.0
Speed-up taken as ratio of scipy.compiler to standard Numeric runs.



# weave and Laplace's equation

Weave case study: An iterative solver for Laplace's Equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$




**PURE PYTHON 2000 SECONDS**

```
for i in range(1, nx-1):
  for j in range(1, ny-1):
    tmp = u[i,j]
    u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
              (u[i, j-1] + u[i, j+1])*dx2)
             / (2.0*(dx2 + dy2))
    diff = u[i,j] - tmp
    err = err + diff**2
```

**Thanks to Prabhu Ramachandran for designing and running this example. His complete write-up is available at:**
**http://www.scipy.org/documentation/weave/weaveperformance**

TNV4 Need to update this link
Travis N. Vaught, 11/11/2004

# weave and Laplace's equation

ENTHOUGHT

## USING NUMERIC 29.0 SECONDS

```
old_u = u.copy() # needed to compute the error.
u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                 (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)
                * dnr_inv
err = sum(dot(old_u - u))
```

## WEAVE.BLITZ 10.2 SECONDS

```
old_u = u.copy() # needed to compute the error.
expr = """ \
       u[1:-1, 1:-1] =  ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                         (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)
                       * dnr_inv
       """
weave.blitz(expr,size_check=0)
err = sum((old_u - u)**2)
```

# weave and Laplace's equation

**WEAVE.INLINE** **4.3 SECONDS**

```
code = """
       #line 120 "laplace.py" (This is only useful for debugging)
       double tmp, err, diff;
       err = 0.0;
       for (int i=1; i<nx-1; ++i) {
         for (int j=1; j<ny-1; ++j) {
           tmp = u(i,j);
           u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2 +
                     (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv;
           diff = u(i,j) - tmp;
           err += diff*diff;
         }
       }
       return_val = sqrt(err);
       """
err = weave.inline(code, ['u','dx2','dy2','dnr_inv','nx','ny'],
                         type_converters = converters.blitz,
                         compiler = 'gcc',
                         extra_compile_args = ['-O3','-malign-double'])
```



# Laplace Benchmarks

| Method | Run Time (sec) | Speed Up |
|---|---|---|
| Pure Python | 1897.0 | ≈ 0.02 |
| Numeric | 29.0 | 1.00 |
| weave.blitz | 10.2 | 2.84 |
| weave.inline | 4.3 | 6.74 |
| weave.inline (fast) | 2.9 | 10.00 |
| Python/Fortran (with f2py) | 3.2 | 9.06 |
| Pure C++ Program | 2.4 | 12.08 |



Debian Linux, Pentium III, 450 MHz, Python 2.1, 192 MB RAM
Laplace solve for 500x500 grid and 100 iterations
Speed-up taken as compared to Numeric

ENTHOUGHT

# f2py

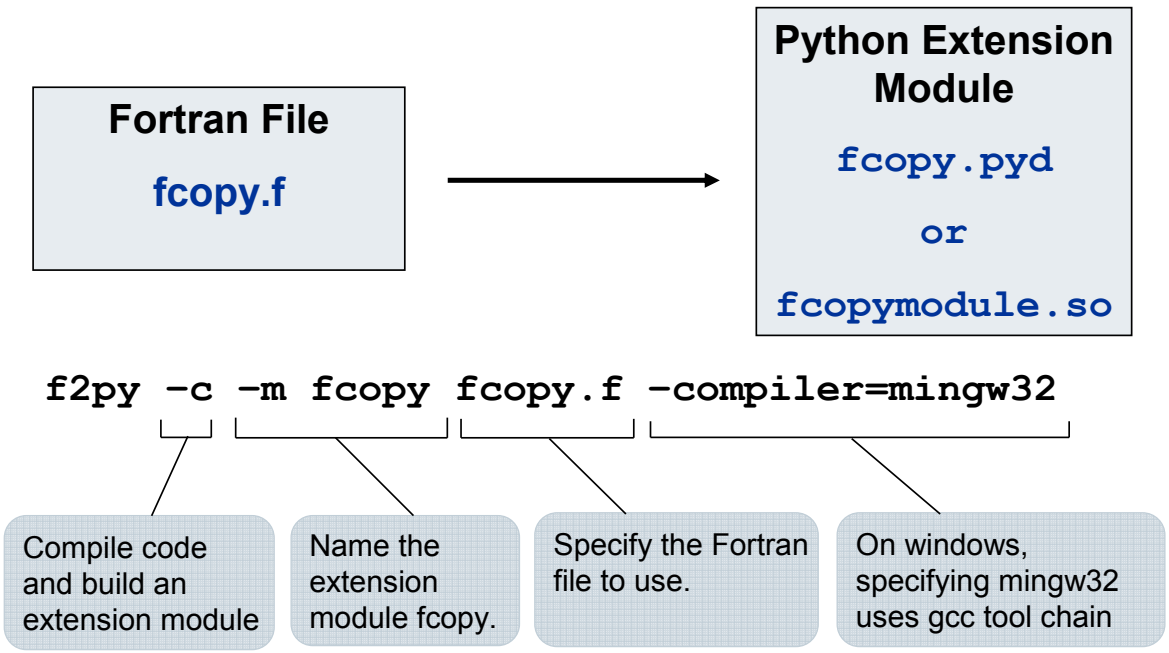


ENTHOUGHT

## f2py

- Author: Pearu Peterson at Center for Nonlinear Studies Tallinn, Estonia
- Automagically "wraps" Fortran 77/90/95 libraries for use in Python. *Amazing.*
- f2py is specifically built to wrap Fortran functions using NumPy arrays.

# Simplest f2py Usage





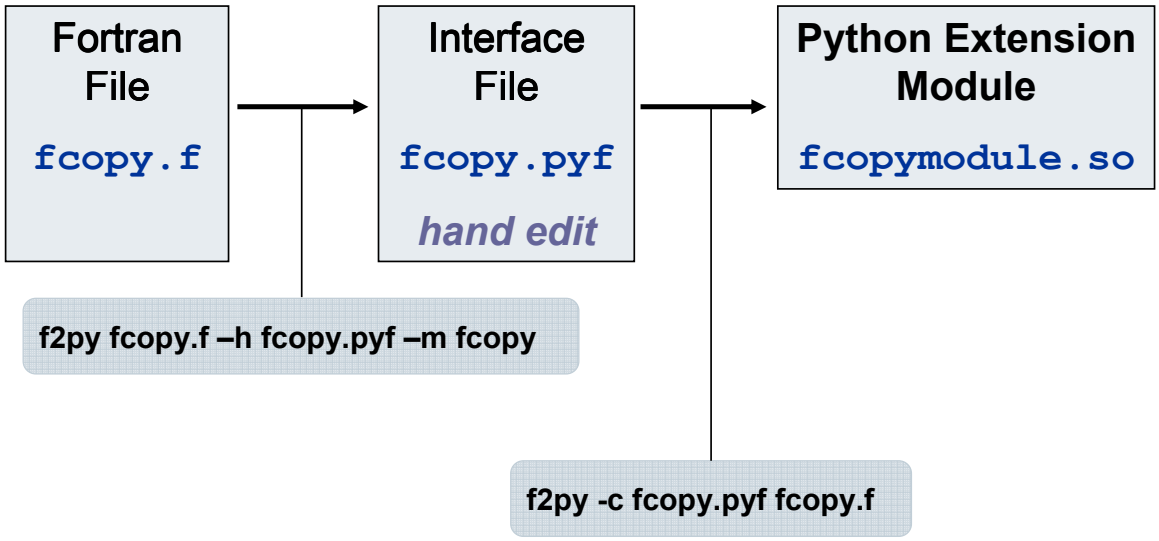

# Simplest Usage Result

```
Fortran file fcopy.f
C
      SUBROUTINE FCOPY(AIN,N,AOUT)
C
      DOUBLE COMPLEX AIN(*)
      INTEGER N
      DOUBLE COMPLEX AOUT(*)
      DO 20 J = 1, N
         AOUT(J) = AIN(J)
20    CONTINUE
      END
```

Looks exactly like the Fortran --- but now in Python!

```
>>> a = rand(1000) + 1j*rand(1000)
>>> b = zeros((1000,), dtype=complex128)
>>> fcopy.fcopy(a,1000,b)
>>> alltrue(a==b)
True
```

# More Sophisticated






# More Sophisticated

Interface file fcopy2.pyf

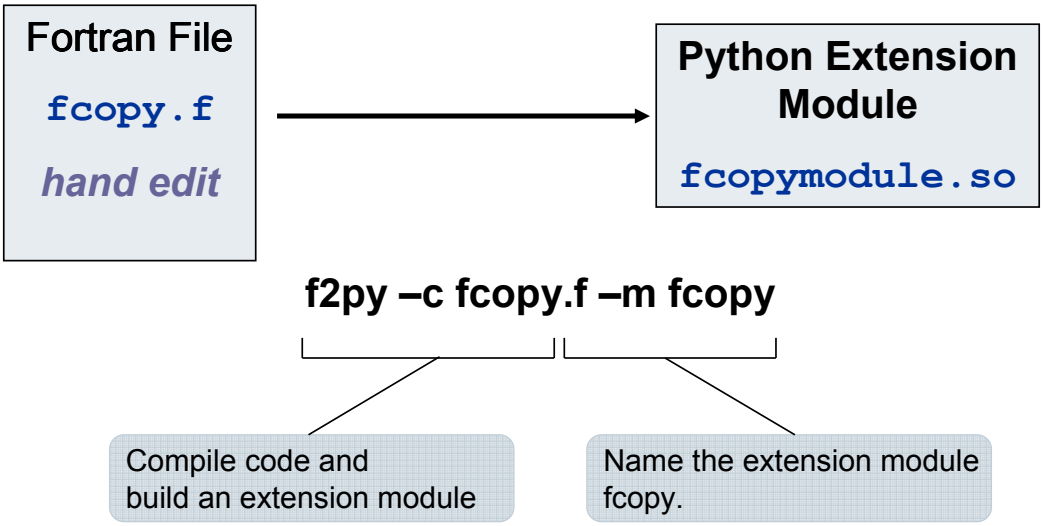
```
!    -*- f90 -*-
python module fcopy2 ! in
    interface  ! in :fcopy
        subroutine fcopy(ain,n,aout) ! in :fcopy:fcopy.f
            double complex dimension(n), intent(in) :: ain
            integer, intent(hide),depend(ain) :: n=len(ain)
            double complex dimension(n),intent(out) :: aout
        end subroutine fcopy
    end interface
end python module fcopy

! This file was auto-generated with f2py (version:2.37.233-1545).
! See http://cens.ioc.ee/projects/f2py2e/
```

Give f2py some hints as to what these variables are used for and how they may be related in Python.

```
fcopy - Function signature:
  aout = fcopy(ain)
Required arguments:
  ain : input rank-1 array('D')
        with bounds (n)
Return objects:
  aout : rank-1 array('D') with
         bounds (n)
```

```
# More pythonic behavior
>>> a = rand(5).astype(complex64)
>>> b = fcopy2.fcopy(a)
>>> alltrue(a==b)
True
# However, b is complex128, not
# 64 because of signature
>>> print b.dtype
dtype('complex128')
```

# Simply Sophisticated






# Simply Sophisticated

```
Fortran file fcopy2.f
C
      SUBROUTINE FCOPY(AIN,N,AOUT)
C
CF2PY INTENT(IN), AIN
CF2PY INTENT(OUT), AOUT
CF2PY INTENT(HIDE), DEPEND(A), N=LEN(A)
      DOUBLE COMPLEX AIN(*)
      INTEGER N
      DOUBLE COMPLEX AOUT(*)
      DO 20 J = 1, N
         AOUT(J) = AIN(J)
 20   CONTINUE
      END
```

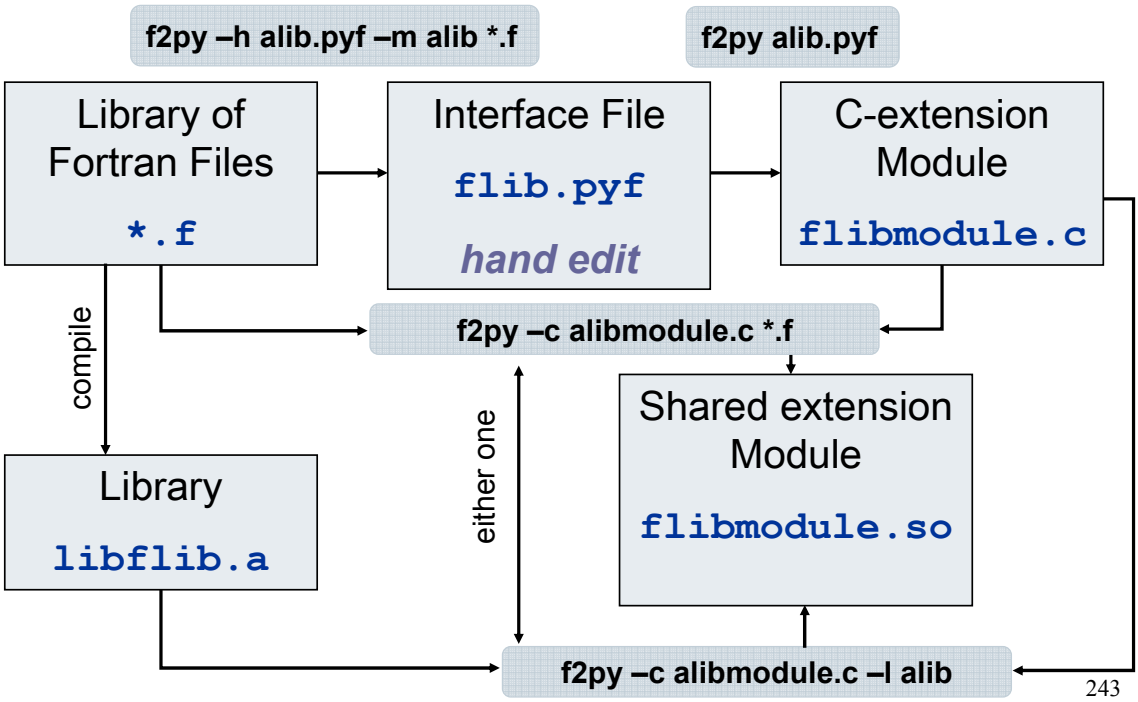
A few directives can help f2py interpret the source.

```
>>> a = rand(1000)
>>> import fcopy
>>> b = fcopy.fcopy(a)
```

Much more Python like!

```
>>> import fcopy
>>> info(fcopy.fcopy)
fcopy - Function signature:
  aout = fcopy(ain)
Required arguments:
  ain : input rank-1 array('D') with
bounds (n)
Return objects:
  aout : rank-1 array('D') with bounds (n)
```

# Saving the Module C-File






# Multidimensional array issues

Python and Numeric use C conventions for array storage (row major order). Fortran uses column major ordering.

Numeric:

A[0,0], A[0,1], A[0,2],…, A[N-1,N-2], A[N-1,N-1]
(last dimension varies the fastest)

Fortran:

A(1,1), A(2,1), A(3,1), …, A(N-1,N), A(N,N)
(first dimension varies the fastest)

f2py handles the conversion back and forth between the representations if you mix them in your code. Your code will be faster, however, if you can avoid mixing the representations (impossible if you are calling out to both C and Fortran libraries that are interpreting matrices differently).

# numpy.distutils

How do I distribute this great new extension module?

- Recipient must have f2py and scipy_distutils installed (both are simple installs)
- Create setup.py file
- Distribute *.f files with setup.py file.
- Optionally distribute *.pyf file if you've spruced up the interface in a separate interface file.

Supported Compilers

- g77, Compaq Fortran, VAST/f90 Fortran, Absoft F77/F90, Forte (Sun), SGI, Intel, Itanium, NAG, Lahey, PG



# Complete Example

In scipy.stats there is a function written entirely in Python

```
>>> help(stats.morestats._find_repeats)
 _find_repeats(arr)

Find repeats in the array and return a list of the
repeats and how many there were.
```

Goal: Write an equivalent fortran function and link it in to Python with f2py so it can be distributed with scipy_base (which uses scipy_distutils) and be available for stats.

Python algorithm uses sort and so we will need a fortran function for that, too.

# The "Decorated" Fortran File

```
Fortran file futil.f
C     Sorts an array arr(1:N) into
      SUBROUTINE DQSORT(N,ARR)
CF2PY INTENT(IN,OUT,COPY), ARR
CF2PY INTENT(HIDE), DEPEND(ARR), N=len(ARR)
      INTEGER N,M,NSTACK
      REAL*8 ARR(N)
      PARAMETER (M=7, NSTACK=100)
      INTEGER I,IR,J,JSTACK, K,L, ISTACK(NSTACK)
      REAL*8 A,TEMP
        …
      END

C     Finds repeated elements of ARR
      SUBROUTINE DFREPS(ARR,N,REPLIST,REPNUM,NLIST)
CF2PY INTENT(IN), ARR
CF2PY INTENT(OUT), REPLIST
CF2PY INTENT(OUT), REPNUM
CF2PY INTENT(OUT), NLIST
CF2PY INTENT(HIDE), DEPEND(ARR), N=len(ARR)
      REAL*8 REPLIST(N), ARR(N)
      REAL*8 LASTVAL
      INTEGER REPNUM(N)
      INTEGER HOWMANY, REPEAT, IND, NLIST, NNUM
         …
      END
```



# setup.py

```
from numpy.distutils.core import Extension

# For f2py extensions that have a .pyf file.
futil1 = Extension(name='futil',
                   sources=['futil.pyf','futil.f'])
# fortran files with f2py directives embedded in them
# are processed by the build_src step of setup.py
futil2 = Extension(name='futil2',
                   sources=['futil2.f'])

if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(name='repeats_util',
          description = "f2py example",
          ext_modules = [futil1, futil2],
          py_modules = 'util.py')
```

**Building:**

```
C:\demo\f2py_repeats> python setup.py build_src \
                      build_ext --inplace --compiler=mingw32
```

**or**

```
C:\demo\f2py_repeats> python setup.py build_src \
                      build_ext --compiler=mingw32 build
```

# The Python "wrapper" function

```
# util.py

import futil2

def find_repeats(arr):
    """Find repeats in arr and return (repeats, repeat_count)
    """
    v1,v2, n = futil2.dfreps(arr)
    return v1[:n],v2[:n]

if __name__ == "__main__":
    from scipy import stats, float64
    a = stats.randint(1, 30).rvs(size=1000)
    print a #.astype(float64)
    repeats, nums = find_repeats(a)
    print 'repeats:'
    print repeats
    print 'nums:'
    print nums
```



# Complete Example

**Try It Out!!**

```
>>> from scipy import stats
>>> from util import find_repeats
>>> a = stats.randint(1,30).rvs(size=1000)
>>> reps, nums = find_repeats(a)
>>> print reps
[  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.
  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.
  23.  24.  25.  26.  27.  28.  29.]
>>> print nums
[29 37 29 30 34 39 46 20 30 32 35 42 40 39 35 26 38 33 40
 29 34 26 38 45 39 38 29 39 29]
```

New function is 25 times faster than the plain Python version

ENTHOUGHT

# SWIG

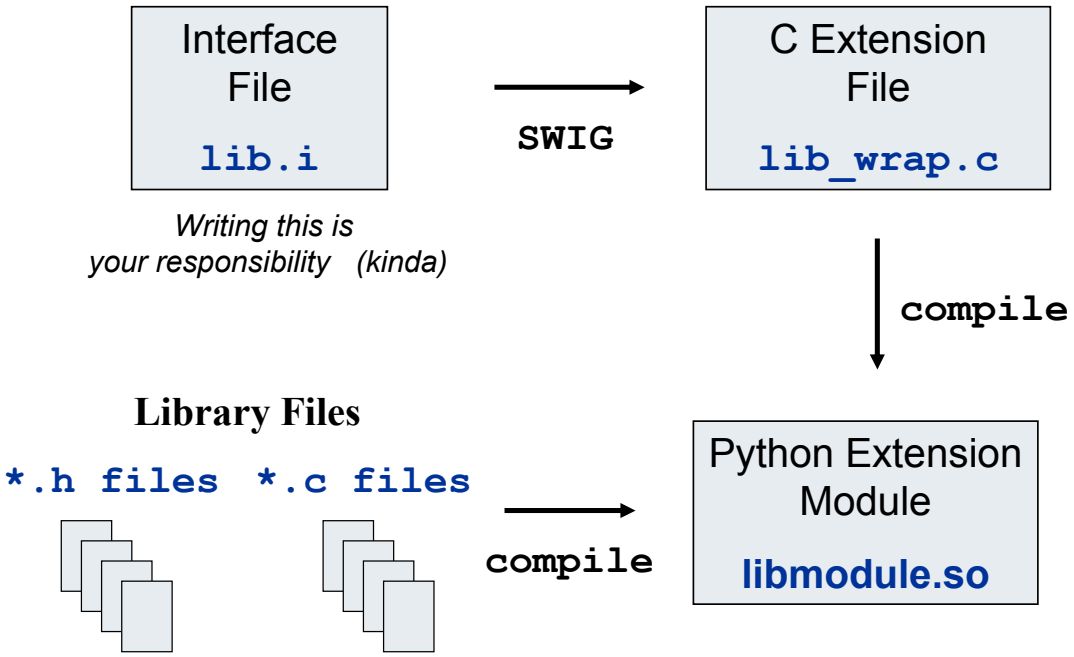


ENTHOUGHT

## SWIG

- Author: David Beazley
- Automatically “wraps” C/C++ libraries for use in Python. *Amazing.*
- SWIG uses interface files to describe library functions
  - No need to modify original library code
  - Flexible approach allowing both simple and complex library interfaces
- Well Documented

# SWIG Process






# Simple Example

**fact.h**

```
#ifndef FACT_H
#define FACT_H

int fact(int n);

#endif
```

**fact.c**

```
#include "fact.h"
int fact(int n)
{
   if (n <=1) return 1;
   else return n*fact(n-1);
}
```

See demo/swig for this example.
Build it using build.bat

**example.i**

```
// Define the modules name
%module example

// Specify code that should
// be included at top of
// wrapper file.
%{
   #include "fact.h"
%}

// Define interface. Easy way
// out - Simply include the
// header file and let SWIG
// figure everything out.
%include "fact.h"
```

# Command Line Build

## LINUX

```
# Create example_wrap.c file
[ej@bull ej]$ swig -python example.i
# Compile library and example_wrap.c code using
# "position independent code" flag
[ej@bull ej]$ gcc -c -fpic example_wrap.c fact.c      \
              -I/usr/local/include/python2.1          \
              -I/usr/local/lib/python2.1/config
# link as a shared library.
[ej@bull ej]$ gcc -shared example_wrap.o fact.o       \
              -o examplemodule.so
# test it in Python
[ej@bull ej]$ python
 ...
 >>> import example
 >>> example.fact(4)
 24
```

For notes on how to use SWIG with VC++ on Windows, see
http://www.swig.org/Doc1.1/HTML/Python.html#n2



# The Wrapper File

## example_wrap.c

```
static PyObject *_wrap_fact(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    int arg0 ;
    int result ;
    /* parse the Python input arguments and extract */

    if(!PyArg_ParseTuple(args,"i:fact",&arg0)) return NULL;

    /* call the actual C function with arg0 as the argument*/
    result = (int )fact(arg0);

    /* Convert returned C value to Python type and return it*/
    resultobj = PyInt_FromLong((long)result);
    return resultobj;
}
```

name of function to return in case of error

first arg in args read into arg0 as int

# SWIG Example 2

**vect.h**

```
int* vect(int x,int y,int z);
int sum(int* vector);
```

**vect.c**

```
#include <malloc.h>
#include "vect.h"
int* vect(int x,int y, int z){
  int* res;
  res = malloc(3*sizeof(int));
  res[0]=x;res[1]=y;res[2]=z;
  return res;
}
int sum(int* v) {
  return v[0]+v[1]+v[2];
}
```

**example2.i**

Identical to example.i if you replace "fact" with "vect".

**TEST IN PYTHON**

```
>>> from example2 import *
>>> a = vect(1,2,3)
>>> sum(a)
6   #works fine!

# Let's take a look at the
# integer array a.
>>> a
'_813d880_p_int'
# WHAT THE HECK IS THIS???
```



# Complex Objects in SWIG

- SWIG treats all complex objects as pointers.
- These C pointers are mangled into string representations for Python's consumption.
- This is one of SWIG's secrets to wrapping virtually any library automatically,
- But… the string representation is pretty primitive and makes it "un-pythonic" to observe/manipulate the contents of the object.
- **Enter typemaps**

# Typemaps

## example_wrap.c

```
static PyObject *_wrap_sum(PyObject *self, PyObject *args) {

  ...
  if(!PyArg_ParseTuple(args,"O:sum",&arg0))
    return NULL;
  ...
  result = (int )sum(arg0);
  ...
  return resultobj;
}
```

Typemaps allow you to insert “type conversion” code into various location within the function wrapper.

Not for the faint of heart. Quoting David:

***“You can blow your whole leg off, including your foot!”***





TNV2 formatting needs fixing
Travis N. Vaught, 11/11/2004

# Typemaps

**The result? Standard C pointers are mapped to NumPy arrays for easy manipulation in Python.**

## YET ANOTHER EXAMPLE – NOW WITH TYPEMAPS

```
>>> import example3
>>> a = example3.vect(1,2,3)
>>> a                    # a should be an array now.
array([1, 2, 3], 'i') # It is!
>>> example3.sum(a)
6
```

The typemaps used for example3 are included in the handouts.

Another example that wraps a more complicated C function used in the previous VQ benchmarks is also provided. It offers more generic handling 1D and 2D arrays.



# Extending Python with C using Pyrex

# What is Pyrex?

ENTHOUGHT

- From the website:
  http://nz.cosc.canterbury.ac.nz/~greg/python/Pyrex

Pyrex lets you write code that mixes Python and C data types any way you want, and compiles it into a C extension for Python.

- The Pyrex language (similar to Python) is written in a separate file (unlike weave)



# Why use Pyrex?

ENTHOUGHT

- Easily access C types
- Much closer to C performance than Python
- Easily access Python types
- Much closer to Python flexibility than C
- Type checking
- Write a Python interface to C code in a Python-like language using Python data types

# A simple Pyrex example

File: pi.pyx

```
def multiply_by_pi( int num ) :
   return num * 3.14159265359
```

File: setup_pi.py

```
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext

setup(
  ext_modules=[ Extension("pi", ["pi.pyx"]) ],
  cmdclass = {'build_ext': build_ext}
)
```

See demo/pyrex for this example.
Build it using build.bat



# A simple Pyrex example

```
C:\> python setup_pi.py build_ext --inplace -c mingw32

C:\> python
Enthought Edition build 1059
Python 2.3.3 (#51, Feb 16 2004, 04:07:52) [MSC v.1200 32
Type "help", "copyright", "credits" or "license" for

>>> import pi

>>> pi.multiply_by_pi()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (0 given)

>>> pi.multiply_by_pi("dsa")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: an integer is required

>>> pi.multiply_by_pi(3)
9.4247779607700011
```

# (some of) the generated code

```
static PyObject *__pyx_f_2pi_multiply_by_pi(PyObject *__pyx_self,
PyObject *__pyx_args, PyObject *__pyx_kwds); /*proto*/
static PyObject *__pyx_f_2pi_multiply_by_pi(PyObject *__pyx_self,
PyObject *__pyx_args, PyObject *__pyx_kwds) {
  int __pyx_v_num;
  PyObject *__pyx_r;
  PyObject *__pyx_1 = 0;
  static char *__pyx_argnames[] = {"num",0};
  if (!PyArg_ParseTupleAndKeywords(__pyx_args, __pyx_kwds, "i",
__pyx_argnames,
&__pyx_v_num)) return 0;

  /* "C:\pi.pyx":2 */
  __pyx_1 = PyFloat_FromDouble((__pyx_v_num * 3.14159265359)); if
(!__pyx_1) {__
pyx_filename = __pyx_f[0]; __pyx_lineno = 2; goto __pyx_L1;}
  __pyx_r = __pyx_1;
  __pyx_1 = 0;
```



# Writing a Pyrex file

## CREATING FUNCTIONS

- If a function is to be called by Python (ie. not from within the .pyx file), it must use **`def`**
- The **`cdef`** keyword defines a function which is private to the .pyx file
- Both **`def`** and **`cdef`** can contain code which freely mix Python and C
- Calls to **`cdef`** functions are faster and sometimes necessary since they have known return types

# Writing a Pyrex file

```
cdef float multiply_by_pi(int num):
    return num * 3.14159265359
```

- This function is only accessible from the .pyx file
- This function can be used by anything needing a float



# Writing a Pyrex file

## ACCESSING OTHER C CODE

- Pyrex needs to know how to call other C routines (cannot simply #include a header)
- Use cdef to create prototypes

```
cdef extern int strlen(char *c)
```

strlen can now be used in Pyrex code...

```
cdef int get_len(char *message):
    return strlen(message)
```

# Writing a Pyrex file

## ACCESSING C STRUCTS

- Pyrex needs to know how to access C structs (again, cannot simply #include a header)
- Use cdef to define a struct (see c_numpy.pxd)

```
cdef extern from "numpy/arrayobject.h":
    ctypedef struct npy_cdouble:
        double real
        double imag
    ...
```

Name of header to include in generated code



# Writing a Pyrex file

## CREATING CLASSES

- Use **`cdef`**, only in this case it will be accessible from Python

File: shrubbery.pyx

```
cdef class Shrubbery:
  cdef int width, height

  def __init__(self, w, h):
      self.width = w
      self.height = h

  def describe(self):
      print "This shrubbery is", \
            self.width, \
            "by", self.height, "cubits."
```

# Writing a Pyrex file

- Use the new class just as if it were written in Python

```
>>> import shrubbery
>>> x = shrubbery.Shrubbery(1, 2)
>>> x.describe()
This shrubbery is 1 by 2 cubits.
>>> print x.width
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: `shrubbery.Shrubbery' object has no
  attribute 'width`
```

- Not truly a Python class though!



# Pyrex in action

## COMPUTING THE LAPLACIAN

File: pyx_lap.pyx

```
from c_numpy cimport import_array, ndarray, \
                     npy_intp, NPY_DOUBLE, \
                     PyArray_ContiguousFromAny
# intialize array module.
import_array()

...
```

# Pyrex in action

File: pyx_lap.pyx (cont.)

```
...
def pyrexTimeStep(object ou, double dx, double dy):
    # convert ou to a contiguous double array with mindim=2 and maxdim=2
    cdef ndarray u
    u = PyArray_ContiguousFromAny(ou, NPY_DOUBLE, 2, 2)

    # Cast the data pointer in the numpy array to a double* pointer
    cdef double *elem
    elem = <double *>u.data

    ...

    for i from 1 <= i < nx-1:
        uc = elem + i*ny + 1
        ...
        for j from 1 <= j < ny-1:
            tmp = uc[0]
            uc[0] = ((ul[0] + ur[0])*dy2 + (uu[0] + ud[0])*dx2)*dnr_inv
            ...
    return sqrt(err)
```



# Pyrex in action

File: laplace.py

```
...
 def pyrexTimeStep(self, dt=0.0):
        """Takes a time step using a function written in Pyrex.  Use
        the given setup.py to build the extension using the command
        python setup.py build_ext --inplace.  You will need Pyrex
        installed to run this."""

        g = self.grid
        err = pyx_lap.pyrexTimeStep(g.u, g.dx, g.dy)
        return err
...
```

# Pyrex in action

```
C:\> python setup.py build_ext --inplace -c mingw32

C:\> python laplace.py

Doing 100 iterations on a 500x500 grid...

        Numeric took 0.37059389964 seconds
        pyrex took 0.370561913397 seconds

pure Python (1 iteration) took 1.53459582054 seconds
100 iterations should take about 153.459582 seconds
```



# Topics (3rd Day)

- Parallel Programming
- 3D Visualization – Mayavi
- 2D Visualization (Part 2…)

# Parallel Programming in Python



## Parallel Computing Tools

- Python has threads (sort'a)
- COW (www.scipy.org)
- pyMPI(pympi.sf.net/)
- pyre (CalTech)
- PyPAR (datamining.anu.edu.au/~ole/pypar/)
- SCIENTIFIC (starship.python.net/crew/hinsen)

- Mpi4py (http://mpi4py.scipy.org/)
- Ipython1 (http://ipython.scipy.org/moin/)

# Cluster Computing with Python

- cow.py
  - Pure Python Approach
  - Easy to Use
  - Suitable for “embarrassingly” parallel tasks
- pyMPI (Message Passing Interface)
  - Developed by Patrick Miller, Martin Casado *et al.* at Lawrence Livermore National Laboratories
  - De-facto industry standard for high-performance computing
  - Vendor optimized libraries on “Big Iron”
  - Possible to integrate existing HPFortran and HPC codes such as *Scalapack* (parallel linear algebra) into Python.



# Threads

- Python threads are built on POSIX and Windows threads (hooray!)
- Python threads share a “lock” that prevents threads from invalid sharing
- Threads pass control to another thread
  - every few instructions
  - during blocking I/O (if properly guarded)
  - when threads die

# The “threading” module

- from threading import Thread
  - a lower level thread library exists, but this is much easier to use
- a thread object can “fork” a new execution context and later be “joined” to another
- you provide the thread body either by creating a thread with a function or by subclassing it



# Making a thread

- we will work at the prompt!

```
>>> from threading import *
>>> def f(): print 'hello'
>>> T = Thread(target=f)
>>> T.start()
```

# Thread operations

- **currentThread**()
- T.start()
- T.join()
- T.getName() / T.setName()
- T.isAlive()
- T.isDaemon()  / T.setDaemon()



# Passing arguments to a thread

```
>>> from threading import *
>>> def f(a,b,c): print 'hello',a,b,c
>>> T = Thread(target=f,args=(11,22),kwargs={'c':3})
>>> T.start()
```

# Subclassing a thread

```
from threading import *
class myThread(Thread):
      def __init__(self,x,**kw):
            Thread.__init__(self,**kw) #FIRST!
            self.x = x
      def run():
            print self.getName()
            print 'I am running',self.x
T = myThread(100)
T.start()
```

NOTE: Only __init__ and run() are available for overload



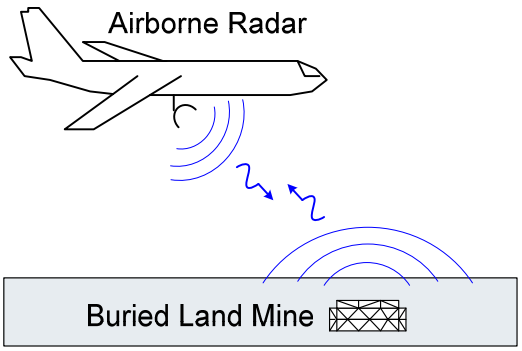
# CAUTION!

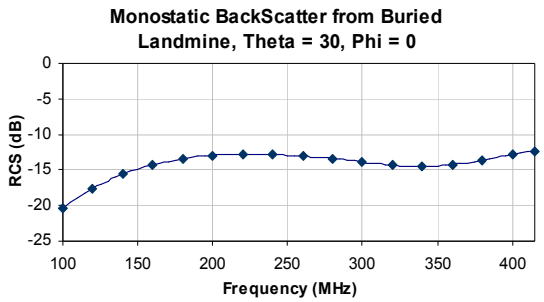
- Only one thread can operate on Python objects at a time
- Internally, threads are switched
- If you write extensions that are intended for threading, use
  - PY_BEGIN_ALLOW_THREADS
  - PY_END_ALLOW_THREADS

# COW



# Electromagnetic Scattering




**Inputs**
environment, target mesh, and *multiple frequencies*

**Mem:** KB to Mbytes

SMALL

**Computation**
$N^3$ CPU
$N^2$ storage
**Time:** a few seconds to days
**Mem:** MB to GBytes

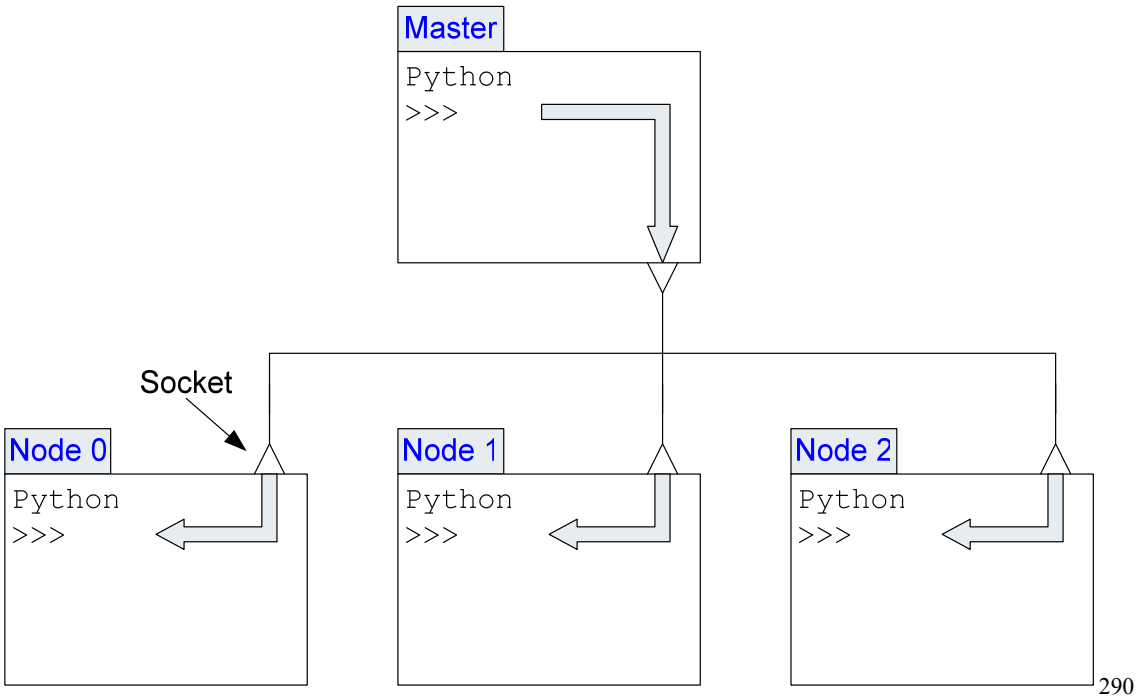
LARGE!

**Outputs**
Radar Cross Section values

**Mem:** KB to MBytes

SMALL

# cow.py






# Cluster Creation

Master

```
>>> import scipy.cow
#                [name, port]
>>> machines = [['s0',11500],['s1',11500],['s2',11500]]
>>> cluster = scipy.cow.machine_cluster(machines)
>>>
```

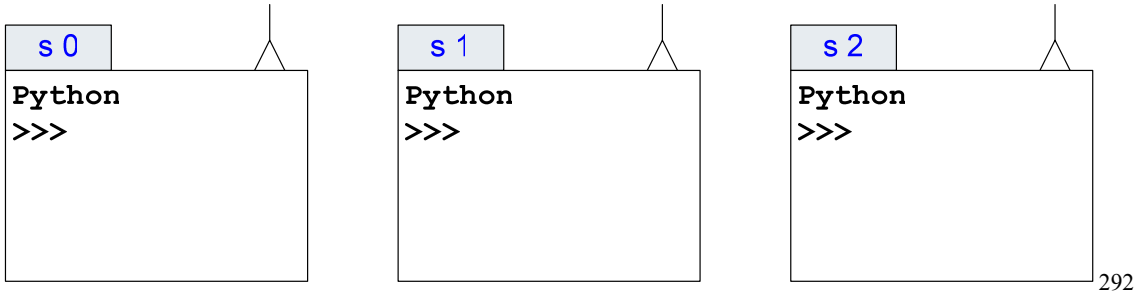
Port numbers below 1024 are reserved by the OS and generally must run as 'root' or 'system'. Valid port numbers are between 1025-49151. Be sure another program is not using the port you choose.

# Starting remote processes

Master

```
>>> cluster = scipy.cow.cluster(machines)
>>> cluster.start()
```

`start()` uses `ssh` to start an interpreter listening on port 11500 on each remote machine

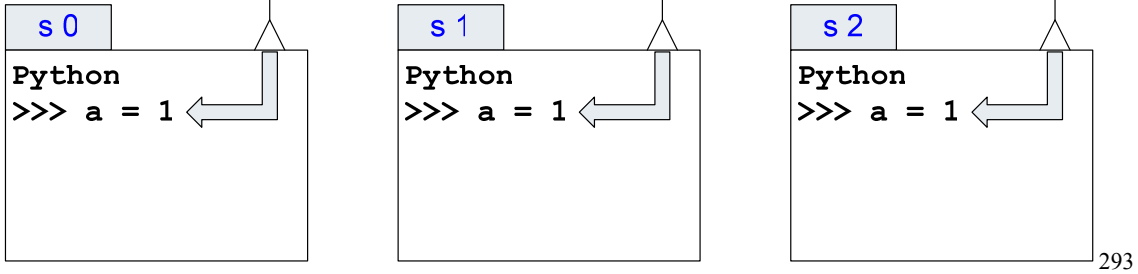




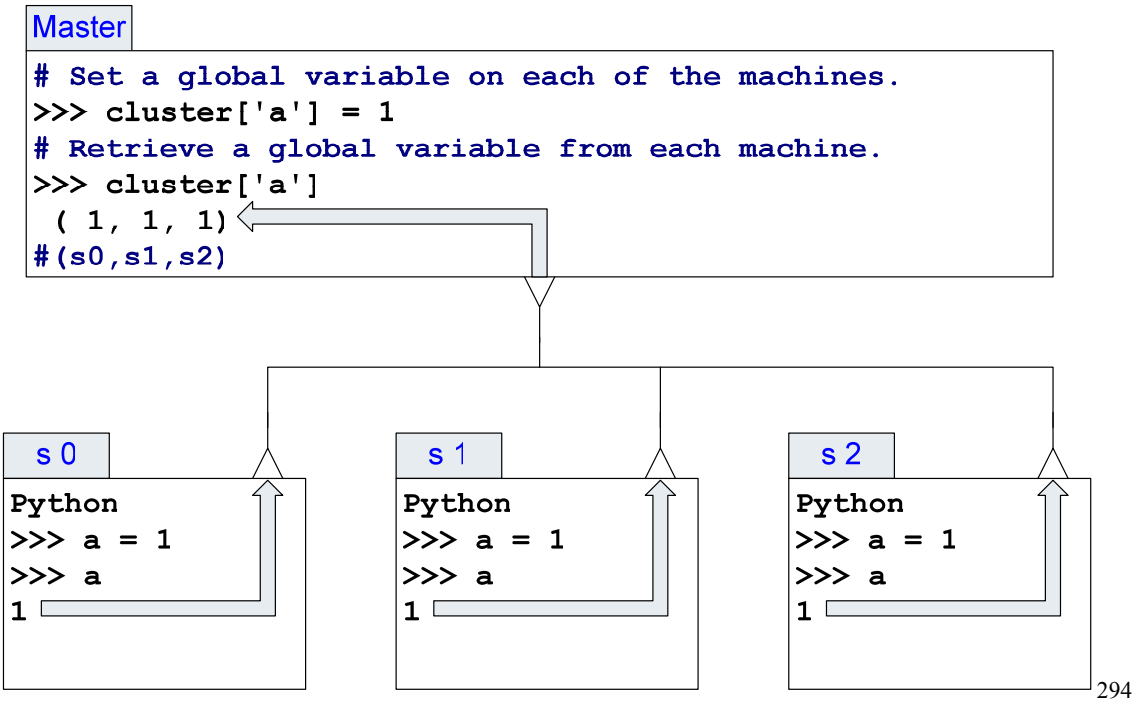

# Dictionary Behavior of Clusters

Master

```
# Set a global variable on each of the machines.
>>> cluster['a'] = 1
```

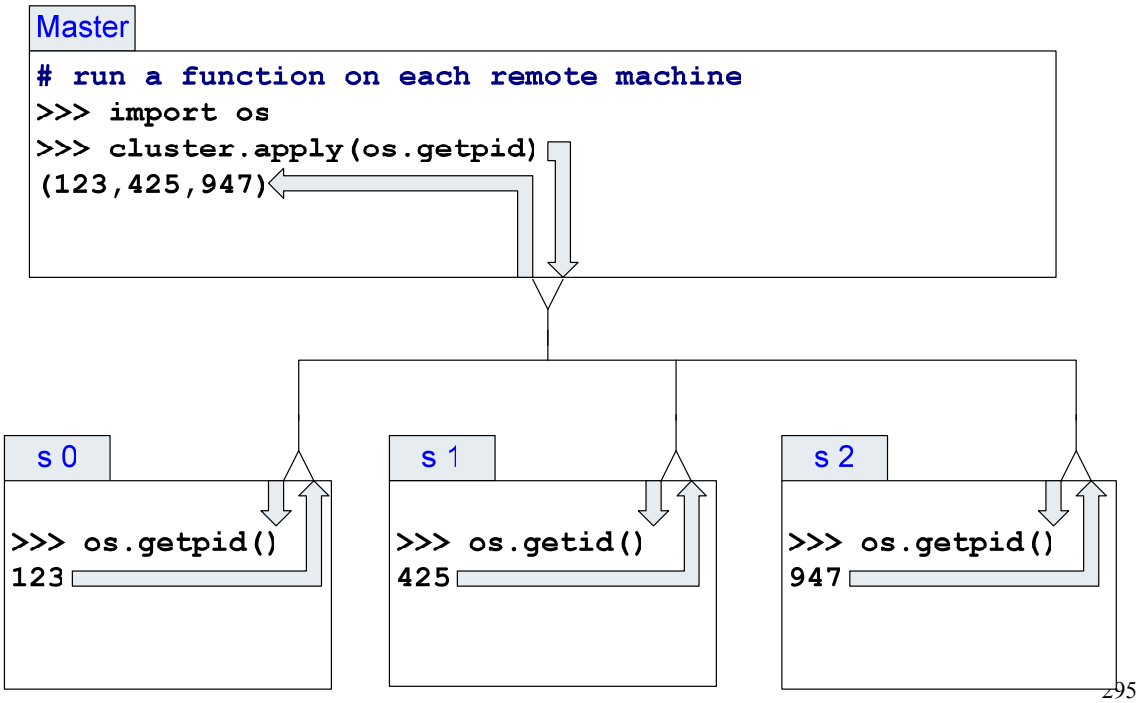


# Dictionary Behavior of Clusters

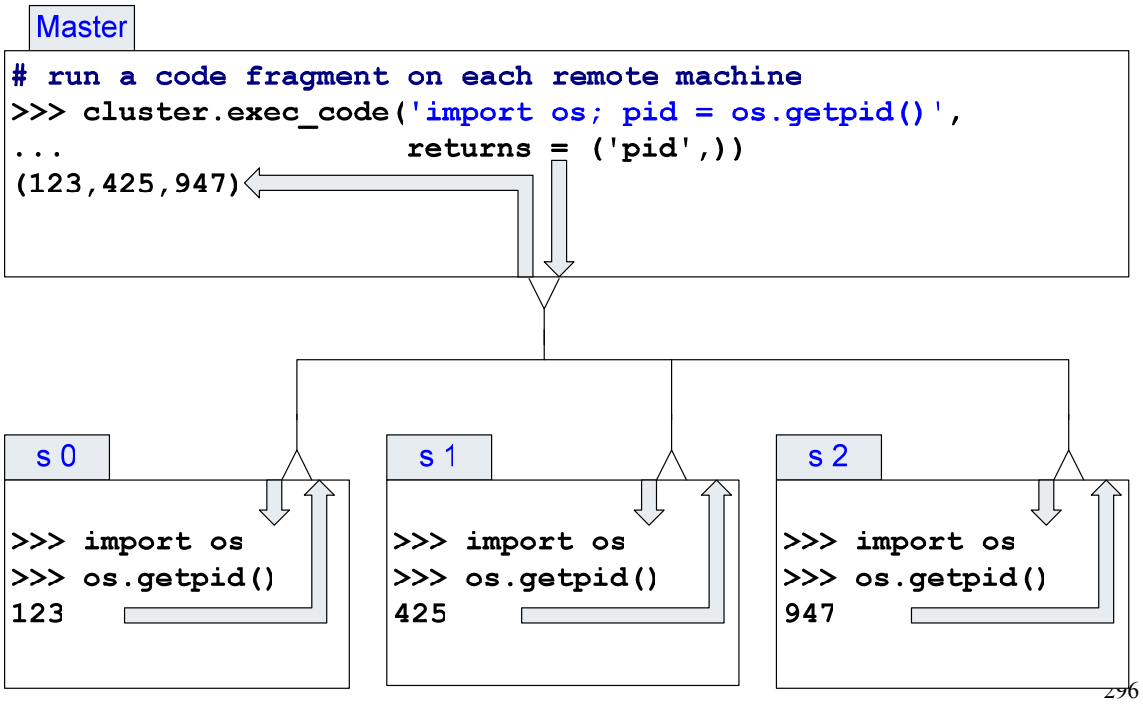





# cluster.apply()

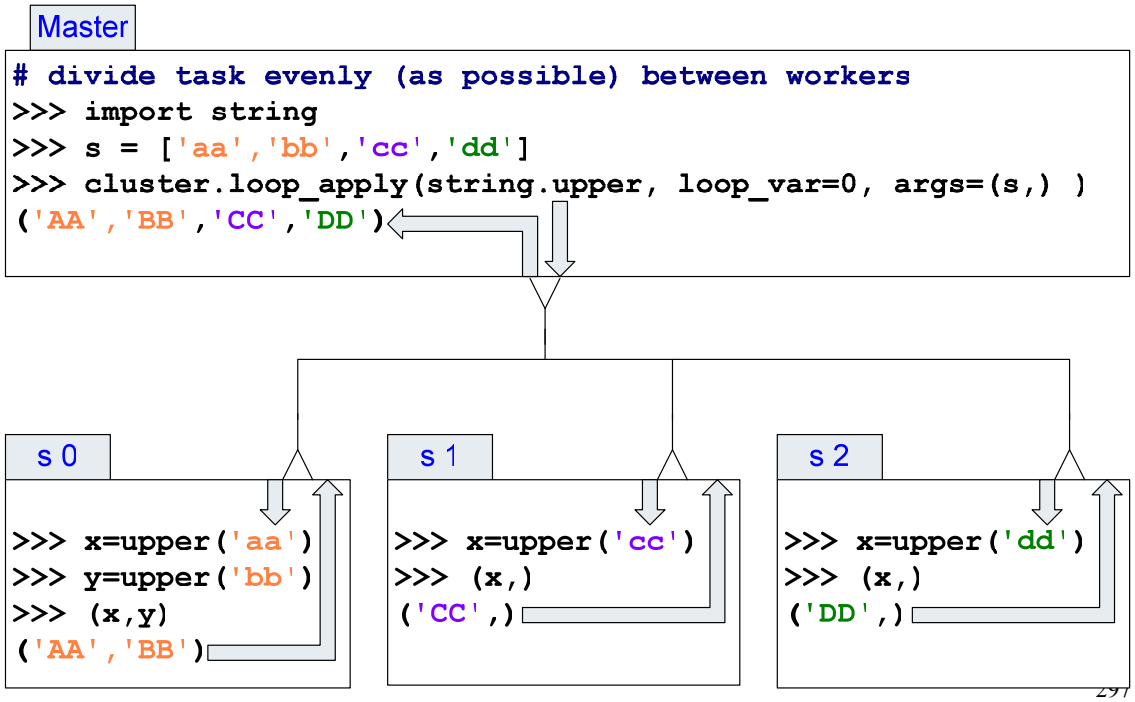


# cluster.exec_code()






# cluster.loop_apply()



# Cluster Method Review

- apply(function, args=(), keywords=None)
  - Similar to Python's built-in apply function. Call the given function with the specified args and keywords on all the worker machines. Returns a list of the results received from each worker.
- exec_code(code, inputs=None, returns=None)
  - Similar to Python's built-in exec statement. Execute the given code on all remote workers as if it were typed at the command line. inputs is a dictionary of variables added to the global namespace on the remote workers. returns is a list of variable names (as strings) that should be returned after the code is executed. If returns contains a single variable name, a list of values is returned by exec_code. If returns is a sequence of variable names, exec_code returns a list of tuples.



# Cluster Method Review

- loop_apply(function,loop_var,args=(), keywords=None)
  - Call function with the given args and keywords. One of the arguments or keywords is actually a sequence of arguments. This sequence is looped over, calling function once for each value in the sequence. loop_var indicates which variable to loop over. If an integer, loop_var indexes the args list. If a string, it specifies a keyword variable. The loop sequence is divided as evenly as possible between the worker nodes and executed in parallel.
- loop_code(code, loop_var, inputs=None, returns=None)
  - Similar to exec_code and loop_apply. Here loop_var indicates a variable name in the inputs dictionary that should be looped over.

# Cluster Method Review

- ps(sort_by='cpu',**filters)
  - Display all the processes running on the remote machine much like the ps Unix command. sort_by indicates which field to sort the returned list. Also keywords allow the list to be filtered so that only certain processes are displayed.
- info()
  - Display information about each worker node including its name, processor count and type, total and free memory, and current work load.



# Query Operations

```
>>> herd.cluster.info()
MACHINE    CPU         GHZ    MB TOTAL   MB FREE    LOAD
s0         2xP3        0.5      960.0      930.0    0.00
s1         2xP3        0.5      960.0       41.0    1.00
s2         2xP3        0.5      960.0      221.0    0.99
```

```
>>> herd.cluster.ps(user='ej',cpu='>50')
MACHINE USER   PID   %CPU   %MEM   TOTAL MB RES MB CMD
s0      ej     123   99.9    0.4    3.836    3.836  python...
s1      ej     425   99.9    0.4    3.832    3.832  python...
s2      ej     947   99.9    0.4    3.832    3.832  python...
```

# Simple FFT Benchmark

## (1) STANDARD SERIAL APPROACH TO 1D FFTs

```
>>> b = fft(a) # a is a 2D array: 8192 x 512
```

## (2) PARALLEL APPROACH WITH LOOP_APPLY

```
>>> b = cluster.loop_apply(fft,0,(a,))
```

## (3) PARALLEL SCATTER/COMPUTE/GATHER APPROACH

```
>>> cluster.import_all('FFT')
# divide a row wise amongst workers
>>> cluster.row_split('a',a)
# workers calculate fft of small piece of a and stores as b.
>>> cluster.exec_code('b=fft(a)')
# gather the b values from workers back to master.
>>> b = cluster.row_gather('b')
```



# FFT Benchmark Results

| Method | CPUs | Run Time (sec) | Speed Up | Efficiency |
|---|---|---|---|---|
| (1) standard | 1 | 2.97 | - | - |
| (2) loop_apply | 2 | 11.91 | 0.25 | -400% |
| (3) scatter/compute/gather | 2 | 13.83 | 0.21 | -500% |

Test Setup:

The array a is 8192 by 512. ffts are applied to each row independently as is the default behavior of the FFT module.

The cluster consists of 16 dual Pentium II 450 MHz machines connected using 100 Mbit ethernet.

# FFT Benchmark Results

| Method | CPUs | Run Time (sec) | Speed Up | Efficiency |
|---|---|---|---|---|
| (1) standard | 1 | 2.97 | - | - |
| (2) loop_apply | 2 | 11.91 | 0.25 | -400% |
| (3) scatter/compute/gather | 2 | 13.83 | 0.21 | -500% |
| (3) compute alone | 2 | 1.49 | 2.00 | 100% |
| (3) compute alone | 4 | 0.76 | 3.91 | 98% |
| (3) compute alone | 16 | 0.24 | 12.38 | 78% |
| (3) compute alone | 32 | 0.17 | 17.26 | 54% |

Moral:

If data can be distributed among the machines once and then manipulated in place, reasonable speed-ups are achieved.



# Electromagnetics

| EM Scattering Problem | CPUs | Run Time (sec) | Speed Up | Efficiency |
|---|---|---|---|---|
| Small Buried Sphere<br>64 freqs, 195 edges | 32 | 8.19 | 31.40 | 98.0% |
| Land Mine<br>64 freqs, 1152 edges | 32 | 285.12 | 31.96 | 99.9% |

# Serial vs. Parallel EM Solver

## SERIAL VERSION

```
def serial(solver,freqs,angles):
    results = []
    for freq in freqs:
        # single_frequency handles calculation details
        res = single_frequency(solver,freq,angles)
        results.append(res)
    return results
```

## PARALLEL VERSION

```
def parallel(solver,freqs,angles,cluster):
    # make sure cluster is running
    cluster.start(force_restart = 0)
    # bundle arguments for loop_apply call
    args = (solver,freqs,angles)
    # looping handled by loop_apply
    results = cluster.loop_apply(single_frequency,1,args)
    return results
```



# pyMPI

# Simple MPI Program

```
# output is asynchronous
% mpirun -np 4 pyMPI
>>> import mpi
>>> print mpi.rank
3
0
2
1
# force synchronization
>>> mpi.synchronizedWrite(mpi.rank, '\n')
0
1
2
3
```



# Broadcasting Data

```
import mpi
import math

if mpi.rank == 0:
    data = [sin(x) for x in range(0,10)]
else:
    data = None

common_data = mpi.bcast(data)
```

# mpi.bcast()

- bcast() broadcasts a value from the “root” process (default is 0) to all other processes
- bcast’s arguments include the message to send and optionally the root sender
- the message argument is ignored on all processors except the root



# Scattering an Array

```
# You can give a little bit to everyone
import mpi
from math import sin,pi
if mpi.rank == 0:
    array = [sin(x*pi/99) for x in range(100)]
else:
   array  = None

# give everyone some of the array
local_array = mpi.scatter(array)
```

# mpi.scatter()

- scatter() splits an array, list, or tuple evenly (roughly) across all processors
- the function result is always a [list]
- an optional argument can change the root from rank 0
- the message argument is ignored on all processors except the root



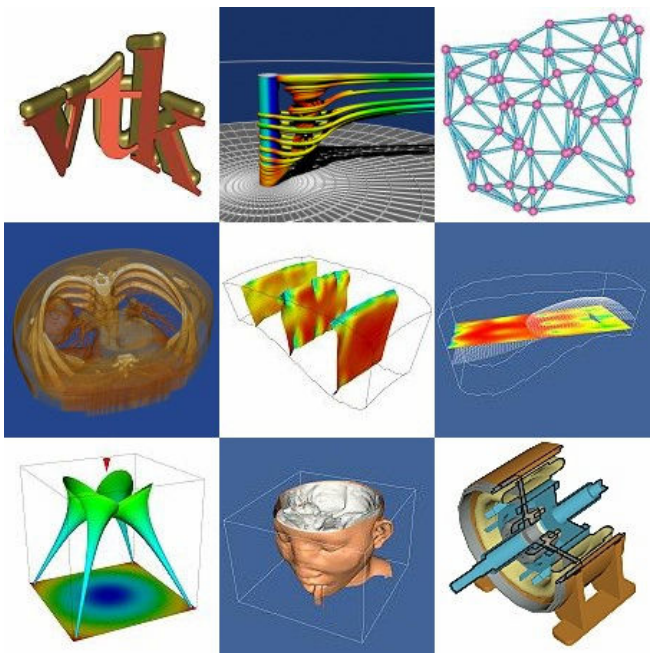
# 3D Visualization with VTK

# Visualization with VTK

- Visualization Toolkit from Kitware
  - www.kitware.com
- Large C++ class library
  - Wrappers for Tcl, Python, and Java
  - Extremely powerful, but…
  - Also complex with a steep learning curve

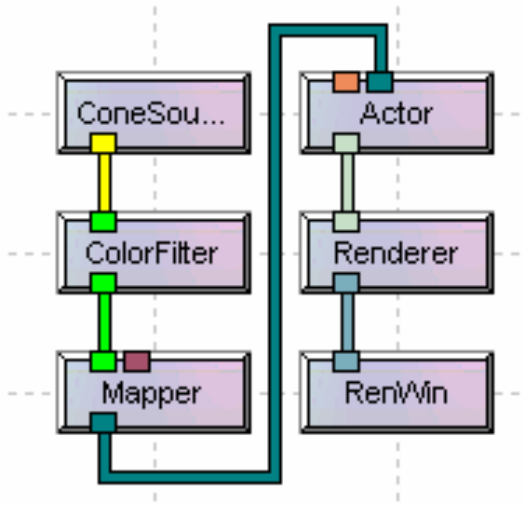


# VTK Gallery



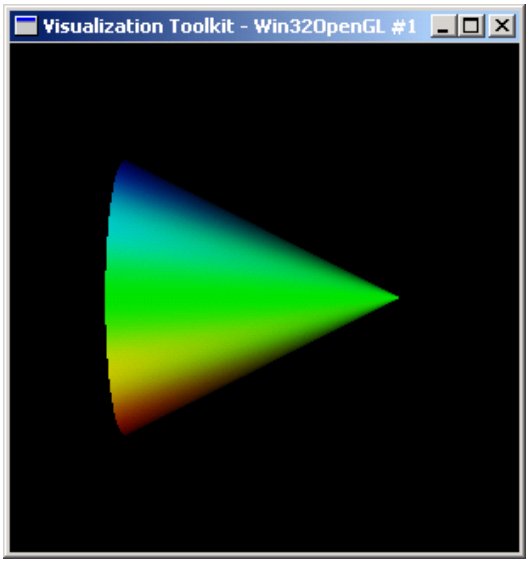
# VTK Pipeline

## PIPELINE




Pipeline view from Visualization Studio at
http://www.principiamathematica.com

## OUTPUT






# Cone Example

## SETUP

eaj2

```
# VTK lives in two modules
from vtk import *

# Create a renderer
renderer = vtkRenderer()

# Create render window and connect the renderer.
render_window = vtkRenderWindow()
render_window.AddRenderer(renderer)
render_window.SetSize(300,300)

# Create Tkinter based interactor and connect render window.
# The interactor handles mouse interaction.
interactor = vtkRenderWindowInteractor()
interactor.SetRenderWindow(render_window)
```

# Cone Example (cont.)

ENTHOUGHT

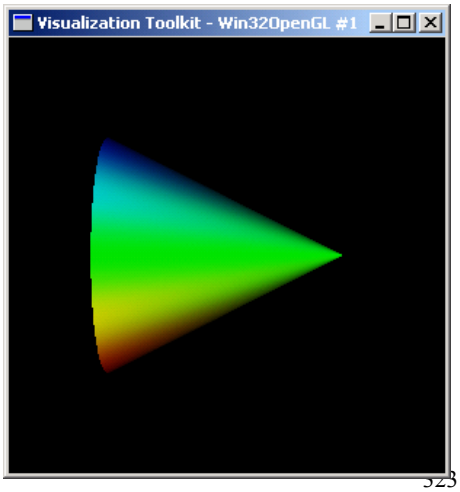
## PIPELINE

```
# Create cone source with 200 facets.
cone = vtkConeSource()
cone.SetResolution(200)

# Create color filter and connect its input
# to the cone's output.
color_filter = vtkElevationFilter()
color_filter.SetInput(cone.GetOutput())
color_filter.SetLowPoint(0,-.5,0)
color_filter.SetHighPoint(0,.5,0)

# map colorized cone data to graphic primitives
cone_mapper = vtkDataSetMapper()
cone_mapper.SetInput(color_filter.GetOutput())
```

# Cone Example (cont.)

## DISPLAY

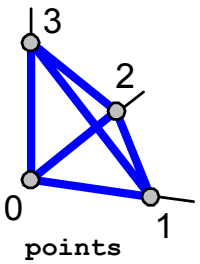
```
# Create actor to represent our
# cone and connect it to the
# mapper
cone_actor = vtkActor()
cone_actor.SetMapper(cone_mapper)

# Assign actor to
# the renderer.
renderer.AddActor(cone_actor)

# Initialize interactor
# and start visualizing.
interactor.Initialize()
interactor.Start()
```






# Mesh Generation

## POINTS AND CELLS




points

| id | x | y | z | temp |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 10 |
| 1 | 1 | 0 | 0 | 20 |
| 2 | 0 | 1 | 0 | 20 |
| 3 | 0 | 0 | 1 | 30 |

triangles

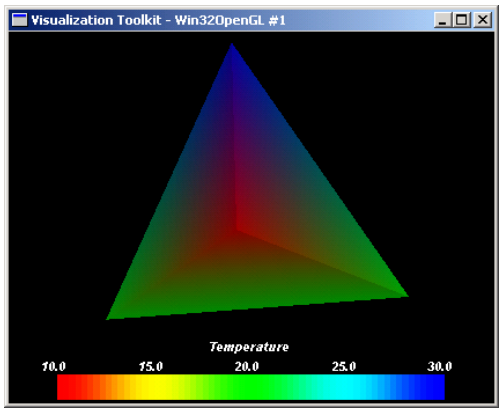
| id | x | y | z |
|---|---|---|---|
| 0 | 0 | 1 | 3 |
| 1 | 0 | 3 | 2 |
| 2 | 1 | 2 | 3 |
| 3 | 0 | 2 | 1 |

```
# Convert list of points to VTK structure
verts = vtkPoints()
temperature = vtkFloatArray()
for p in points:
    verts.InsertNextPoint(p[0],p[1],p[2])
    temperature.InsertNextValue(p[3])


# Define triangular cells from the vertex
# "ids" (index) and append to polygon list.
polygons = vtkCellArray()
for tri in triangles:
    cell = vtkIdList()
    cell.InsertNextId(tri[0])
    cell.InsertNextId(tri[1])
    cell.InsertNextId(tri[2])
    polygons.InsertNextCell(cell)
```

# Mesh Generation

## POINTS AND CELLS

```
# Create a mesh from these lists
mesh = vtkPolyData()
mesh.SetPoints(verts)
mesh.SetPolys(polygons)
mesh.GetPointData().SetScalars( \
...                     temperature)

# Create mapper for mesh
mapper = vtkPolyDataMapper()
mapper.SetInput(mesh)

# If range isn't set, colors are
# not plotted.
mapper.SetScalarRange( \
...         temperature.GetRange())
```

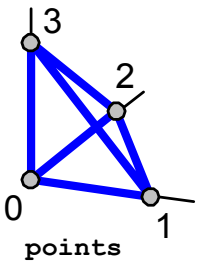



Code for temperature bar not shown.



# TVTK – Simplified VTK interface

## PREVIOUS EXAMPLE USING TVTK – MUCH LESS CODE




points

| id | x | y | z | temp |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 10 |
| 1 | 1 | 0 | 0 | 20 |
| 2 | 0 | 1 | 0 | 20 |
| 3 | 0 | 0 | 1 | 30 |

triangles

| id | x | y | z |
|---|---|---|---|
| 0 | 0 | 1 | 3 |
| 1 | 0 | 3 | 2 |
| 2 | 1 | 2 | 3 |
| 3 | 0 | 2 | 1 |

```
# Convert list of points to VTK structure
points = p[:,:3]
temp = p[:,-1]
traingles = t[:,1:]

# Create mapper for mesh
mesh = vtkPolyData()
mesh.points = points
mesh.polys = triangles
mesh.point_data.scalars = temp

# Create mapper for mesh
mapper = vtkPolyDataMapper()
mapper.input = mesh
mapper.scalar_range = amin(temp),amax(temp)
```

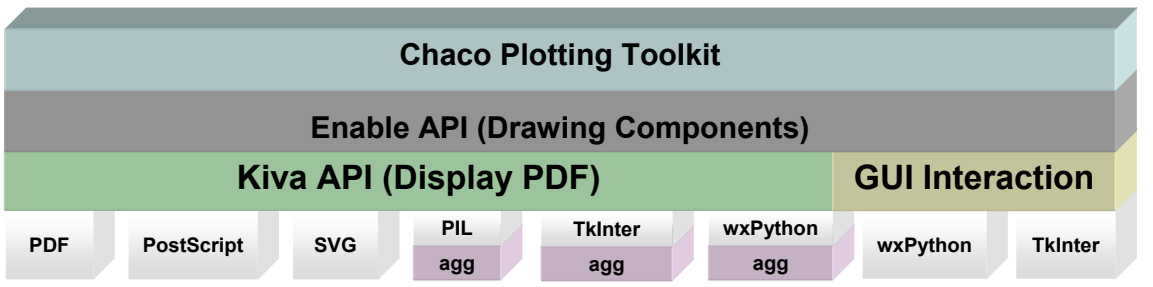
# Chaco Plotting

Eric Jones
eric@enthought.com

Enthought, Inc.
www.enthought.com



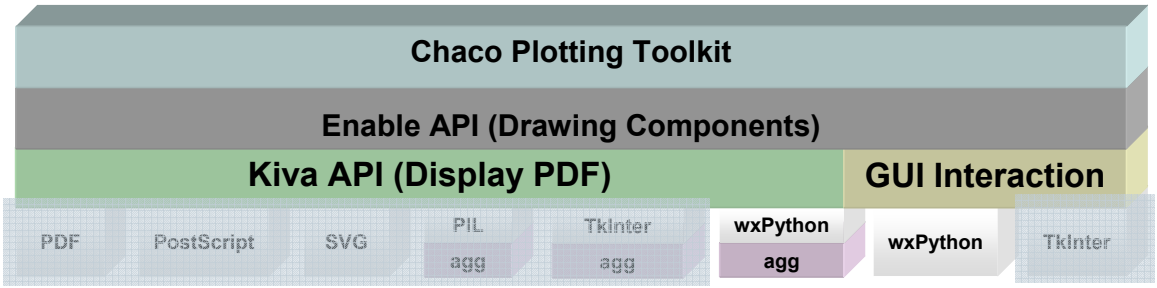
# API Layers




Antigrain Path/Image Renderer
www.antigrain.com

Freetype Font Handling/Rendering
www.freetype.org

# API Layers



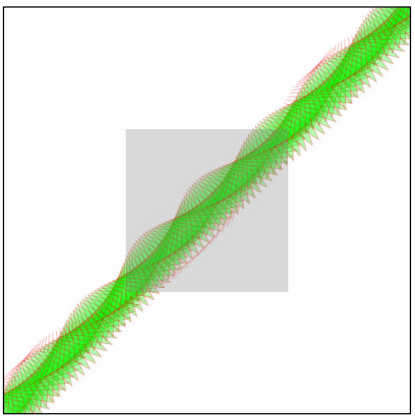

Antigrain Path/Image Renderer
www.antigrain.com

Freetype Font Handling/Rendering
www.freetype.org



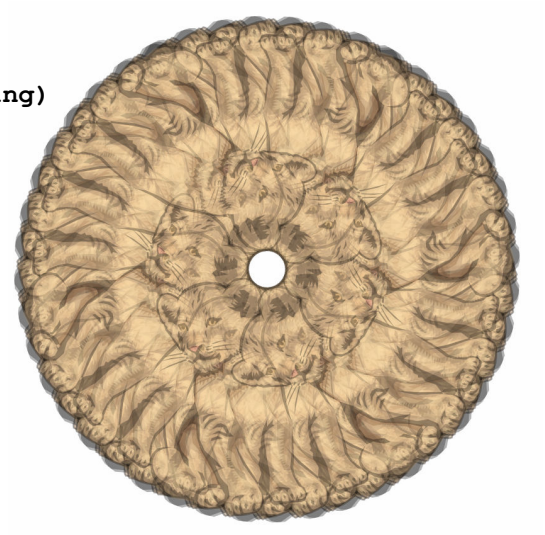
# Kiva -- Transparency

```
gc = agg.GraphicsContextArray((500,500))

gc.set_alpha(0.3)
gc.set_stroke_color((1.0,0.0,0.0))
gc.set_fill_color((0.0,1.0,0.0))

#rotating star pattern
for i in range(0,600,5):
    gc.save_state()
    gc.translate_ctm(i,i)
    gc.rotate_ctm(i*pi/180.)
    add_star(gc)
    gc.draw_path()
    gc.restore_state()

# grey rectangle
gc.set_fill_color((0.5,0.5,0.5))
gc.rect(150,150,200,200)
gc.fill_path()
gc.save("star2.bmp")
```



# Lion – compiled_path

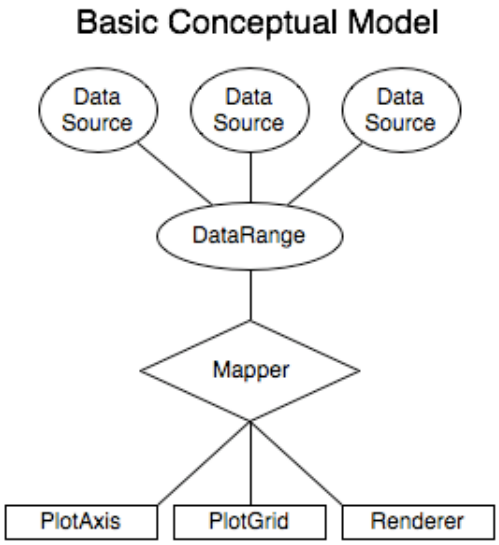
```
Nimages = 90
sz = (1000,1000)
path_and_color = build_paths(lion_string)

gc = agg.GraphicsContextArray(sz)
gc.set_alpha(0.3)
gc.translate_ctm(sz[0]/2.,sz[1]/2.)

for i in range(Nimages):
    for path,color in path_and_color:
        gc.begin_path()
        gc.add_path(path)
        gc.set_fill_color(color)
        gc.fill_path()
    gc.rotate_ctm(1)

bmp = gc.bitmap
agg.save('lion.bmp')
```



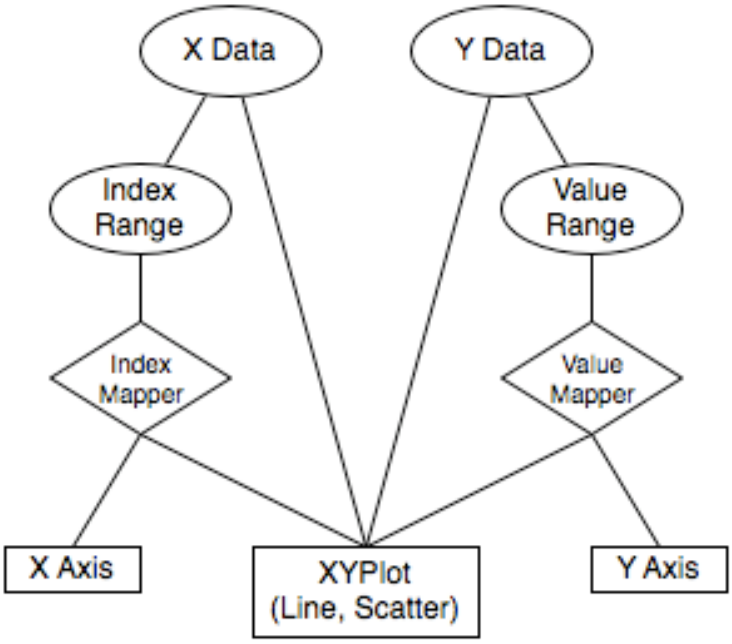
Approximate 100 lions per second on Pentium4 2.2GHz



# Chaco Model



# Chaco Model


X Data
Y Data
Index Range
Value Range
Index Mapper
Value Mapper
X Axis
XYPlot (Line, Scatter)
Y Axis