

Introduction to Scientific Computing with Python

Eric Jones

eric@enthought.com

Enthought

www.enthought.com

Travis Oliphant

oliphant@ee.byu.edu

Brigham Young University

<http://www.ee.byu.edu/>

Modifications by Christos Siopis (IAA, ULB)

Christos.Siopis@ulb.ac.be

Topics

- Introduction to Python
- Numeric Computing
- Basic 2D Visualization
- SciPy (very briefly)

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.

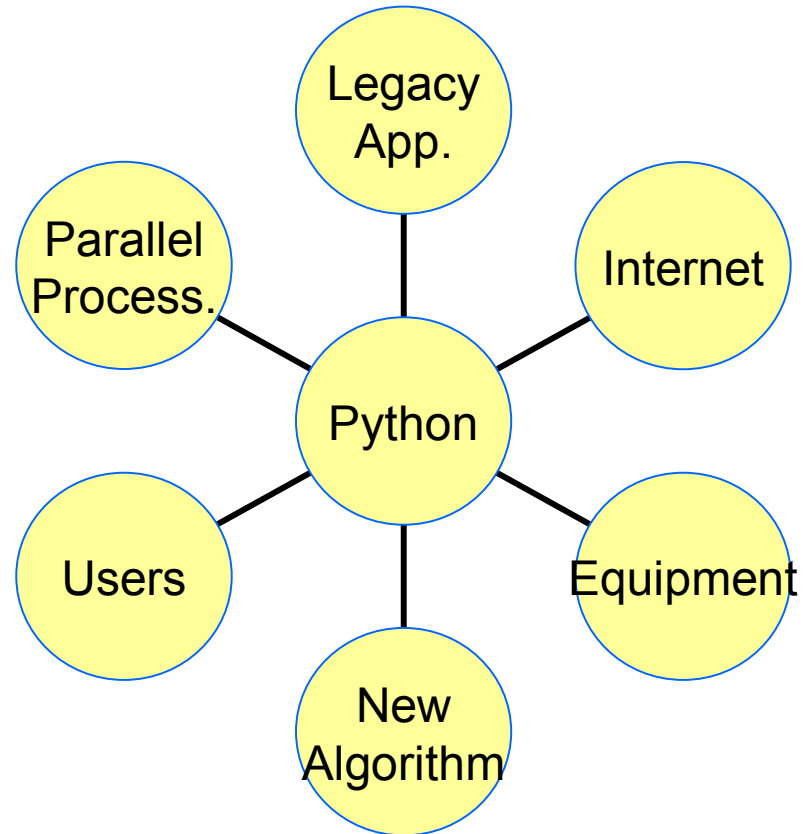
PYTHON HIGHLIGHTS

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

Why Python for glue?

- Python reads almost like “pseudo-code” so it’s easy to pick up old code and understand what you did.
- Python has high-level data structures like lists, dictionaries, strings, and arrays all with useful methods.
- Python has a large module library (“batteries included”) and common extensions covering internet protocols and data, image handling, and scientific analysis.
- Python development is 5-10 times faster than C/C++ and 3-5 times faster than Java

How is Python glue?



Why is Python good glue?

- Python can be embedded into any C or C++ application
Provides your legacy application with a powerful scripting language instantly.
- Python can interface seamlessly with Java
 - Jython www.jython.org
 - JPE jpe.sourceforge.net
- Python can interface with critical C/C++ and Fortran subroutines
 - Rarely will you need to write a main-loop again.
 - Python does not directly call the compiled routines, it uses interfaces (written in C or C++) to do it --- the tools for constructing these interface files are fantastic (sometimes making the process invisible to you).

Tools

- C/C++ Integration

- SWIG www.swig.org

- SIP www.riverbankcomputing.co.uk/sip/index.php

- Pyrex nz.cosc.canterbury.ac.nz/~greg/python/Pyrex

- boost www.boost.org/libs/python/doc/index.html

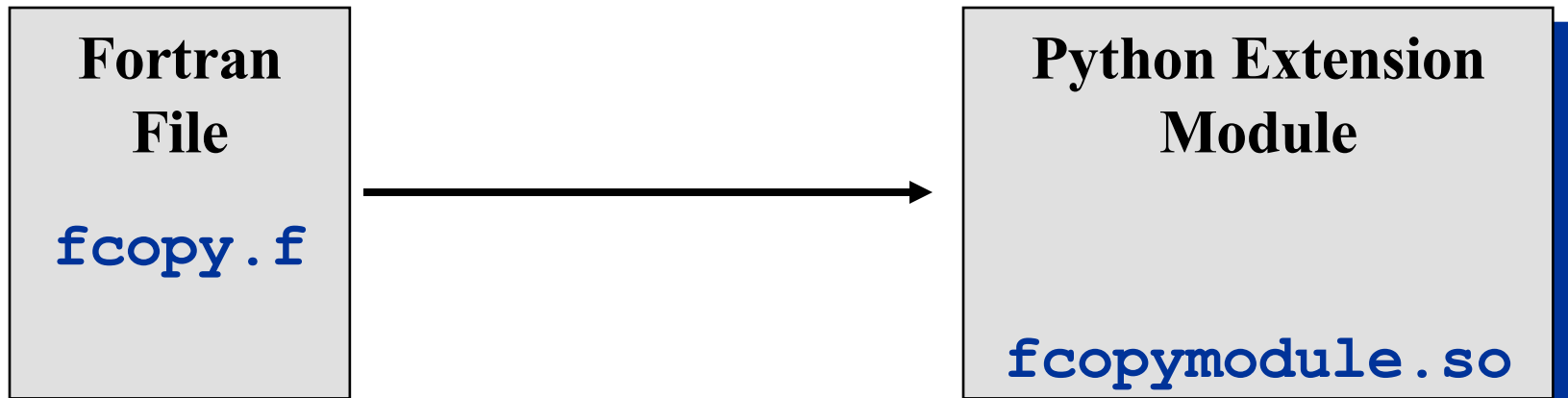
- weave www.scipy.org/site_content/weave

- FORTRAN Integration

- f2py cens.ioc.ee/projects/f2py2e/

- PyFort pyfortran.sourceforge.net

Simplest f2py Usage



```
f2py -c fcopy.f -m fcopy
```

Compile code and build an extension module

Name the extension module fcopy.

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
```

```
>>> import fcopy
>>> info(fcopy)
This module 'fcopy' is auto-generated with f2py
(version:2.37.233-1545).
Functions:
  fcopy(ain,n,aout)
>>> info(fcopy.fcopy)
fcopy - Function signature:
  fcopy(ain,n,aout)
Required arguments:
  ain : input rank-1 array('D') with bounds (*)
  n   : input int
  aout: input rank-1 array('D') with bounds (*)
```

```
>>> a = rand(1000) +
1j*rand(1000)

>>> b = zeros((1000,),'D')

>>> fcopy.fcopy(a,1000,b)
```

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

Who is using Python?

NATIONAL SPACE TELESCOPE LABORATORY

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

INDUSTRIAL LIGHT AND MAGIC

Digital Animation

PAINT SHOP PRO 8

Scripting Engine for JASC
PaintShop Pro 8 photo-editing software

CONOCOPHILLIPS

Oil exploration tool suite

GOOGLE

LAWRENCE LIVERMORE NATIONAL LABORATORIES

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

WALT DISNEY

Digital animation development environment.

REDHAT

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

ENTHOUGHT

Geophysics and Electromagnetics engine scripting, algorithm development, and visualization

Language Introduction

Interactive Calculator

```

$ python
Python 2.4.3 .....
# adding two values
>>> 1 + 1
2
# setting a variable
>>> a = 1
>>> a
1
# checking a variables type
>>> type(a)
<type 'int'>
# an arbitrarily long integer
>>> a = 1203405503201
>>> a
1203405503201L
>>> type(a)
<type 'long'>

```

```

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

```

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

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

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

Strings

CREATING STRINGS

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

STRING OPERATIONS

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

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

STRING LENGTH

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

FORMAT STRINGS

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

The string module

```
>>> import string
>>> s = "hello world"

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

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

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

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

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

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

# strip whitespace from string
>>> s = "\t hello \n"
>>> string.strip(s)
'hello'

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

Multi-line Strings

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

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

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

List objects

LIST CREATION WITH BRACKETS

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

CONCATENATING LIST

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

REPEATING ELEMENTS IN LISTS

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

range(start, stop, step)

```
# the range method is helpful
# for creating a sequence
```

```
>>> range(5)
[0, 1, 2, 3, 4]
```

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

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


Indexing

RETRIEVING 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
```

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]

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

SLICING LISTS

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

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

OMITTING INDICES

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

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

A few methods for list objects

`some_list.append(x)`

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

`some_list.count(x)`

Count the number of times `x` occurs in the list.

`some_list.index(x)`

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

`some_list.remove(x)`

Delete the first occurrence of `x` from the list.

`some_list.reverse()`

Reverse the order of elements in the list.

`some_list.sort(cmp)`

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

List methods in action

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

```
# add an element to the list
```

```
>>> l.append(11)
```

```
>>> print l
```

```
[10,21,23,11,24,11]
```

```
# how many 11s are there?
```

```
>>> l.count(11)
```

```
2
```

```
# where does 11 first occur?
```

```
>>> l.index(11)
```

```
3
```

```
# remove the first 11
```

```
>>> l.remove(11)
```

```
>>> print l
```

```
[10,21,23,24,11]
```

```
# sort the list
```

```
>>> l.sort()
```

```
>>> print l
```

```
[10,11,21,23,24]
```

```
# reverse the list
```

```
>>> l.reverse()
```

```
>>> print l
```

```
[24,23,21,11,10]
```

Mutable vs. Immutable

MUTABLE OBJECTS

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

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



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

IMMUTABLE OBJECTS

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

```
# try inserting values into  
# a string
```

```
>>> s = 'abcde'  
>>> s[1:3] = 'xy'
```

```
Traceback (innermost last):  
File "<interactive input>",line 1,in ?  
TypeError: object doesn't support  
slice assignment
```

```
# here's how to do it
```

```
>>> s = s[:1] + 'xy' + s[3:]  
>>> print s  
'axyde'
```

Tuples

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

TUPLE EXAMPLE

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

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

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]
```

```
# make y a new list equal to x
```

```
>>> y = x[:]
```

x

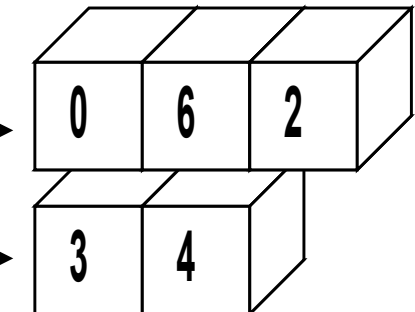
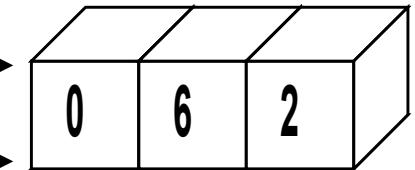
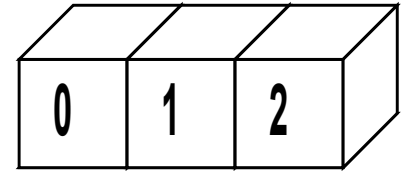
y

x

y

x

y



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.

Indentation is *not* optional but part of the syntax!

IF STATEMENT FORMAT

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

IF EXAMPLE

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

Test Values

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

EMPTY OBJECTS

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

For loops

For loops iterate over a sequence of objects.

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

TYPICAL SCENARIO

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

LOOPING OVER A STRING

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

LOOPING OVER A LIST

```
>>> l=['dogs', 'cats', 'bears']  
>>> accum = ''  
>>> for item in l:  
...     accum = accum + item  
...     # or: 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 True:  
...     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

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.*

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

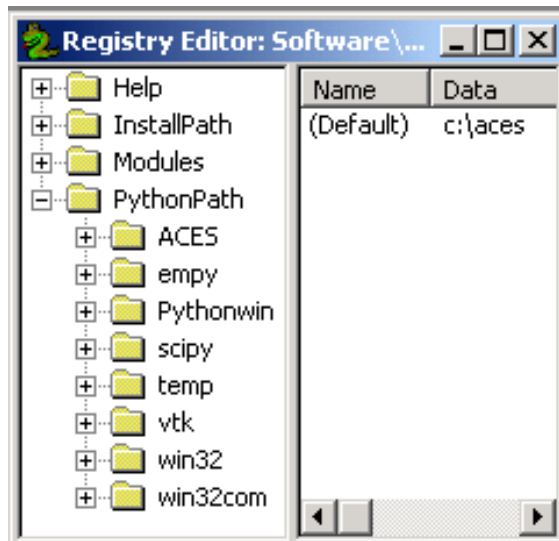
```
def add(x,y):  
    " Add two values."  
    a = x + y  
    return a  
  
def test():  
    l = [0,1,2,3]  
    assert( sum(l) == 6)  
    print 'test passed'  
  
# this code runs only if this  
# module is the main program  
if __name__ == '__main__':  
    test()
```

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/aces
```

UNIX -- .bashrc

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

Setting up PYTHONSTARTUP

PYTHONPATH is an environment variable (or set of registry entries on Windows) that points to a Python file to be executed every time we start the Python *shell*.

UNIX -- .cshrc

```
setenv PYTHONSTARTUP $HOME/bin/py-modules/python-startup.py
```

UNIX -- .bashrc

```
setenv PYTHONSTARTUP $HOME/bin/py-modules/python-startup.py
```

EXAMPLE python-startup.py FILE

```
print 'from numpy import *'  
from numpy import *
```

```
print 'import mp'  
import mp
```

Classes

SIMPLE PARTICLE CLASS

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

EXAMPLE

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


Reading files

FILE INPUT EXAMPLE

```
>>> results = []
>>> f = open('/home/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

More compact version

ITERATING ON A FILE AND LIST COMPREHENSIONS

```
>>> results = []
>>> f = open('/home/rcs.txt', 'r')
>>> f.readline()
'#freq (MHz)  vv (dB)  hh (dB)\n'
>>> for line in f:
...     all = [ float(val) for val in line.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 line.split() ]  
...         for line in open("/home/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

```
import numpy as N
```

```
def rfn(FName, CommentCharacter='#', Type=float, CheckFile=True):  
    Rows = []  
    for line in open(FName, 'r'):  
        if CheckFile:  
            # Truncate line if CommentCharacter is present  
            # (always truncate '\n' in the end when line.find returns -1)  
            line = line[:line.find(CommentCharacter)]  
  
            # Skip empty lines  
            if not line.strip(): continue  
  
            # Split line in words, then convert to Type  
            Rows.append( [Type(x) for x in line.split()] )  
  
            # Make sure all Rows have the same number of columns!  
            lengths = [len(x) for x in Rows]  
            if lengths.count(lengths[0]) != len(lengths):  
                raise RuntimeError, 'mp.rfn(): Rows in file ' + FName + \  
                    ' contain variable number of columns!'  
  
    return N.array(Rows)
```

Exception Handling

ERROR ON LOG OF ZERO

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

CATCHING ERROR AND CONTINUING

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

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

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
```

```
>>> x = [particle(1.2,3.4),particle(2.1,2.3),particle(4.6,.7)]  
>>> x.sort(by_mass)  
>>> x  
[(m:1.2, v:3.4), (m:2.1, v:2.3), (m:4.6, v:0.7)]  
>>> x.sort(by_velocity)  
>>> x  
[(m:4.6, v:0.7), (m:2.1, v:2.3), (m:1.2, v:3.4)]  
>>> x.sort(by_momentum)  
>>> x  
[(m:4.6, v:0.7), (m:1.2, v:3.4), (m:2.1, v:2.3)]
```


Show:

Brief Tour of the Standard Library I & II

(Chapters 10 & 11 of Python Tutorial)

Numpy

Numpy

- Offers Matlab/IDL-ish capabilities within Python
- Web Site
 - <http://www.scipy.org/NumPy>
- Developers (initial coding by Jim Hugunin)
 - Paul Dubouis
 - Travis Oliphant
 - Konrad Hinsien
 - Many more...

Numarray (nearing stable) is optimized for large arrays.

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

Array Operations

IMPORT NUMERIC

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

SIMPLE ARRAY MATH

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

Numeric defines the following constants:



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

MATH FUNCTIONS

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

```
# multiply entire array by
# scalar value
```

```
>>> a = (2*pi)/10.
>>> a
0.628318530718
>>> a*x
array([ 0., 0.628, ..., 6.283])
```

```
# apply functions to array.
```

```
>>> y = sin(a*x)
>>> y
array([ 0.00000000e+00,
        5.87785252e-01,
        ...,
        -2.44929360e-16])
```

Introducing Numeric Arrays

MULTI-DIMENSIONAL ARRAYS

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

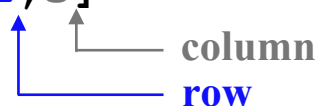
```
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

(ROWS,COLUMNS)

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

GET/SET ELEMENTS

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



```
>>> a[1,3] = -1
```

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

- Show:
 - Numpy Tutorial (HTML)
 - Numpy Example List (HTML)
 - Matplotlib Demo
 - Masked Arrays

Array Slicing

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

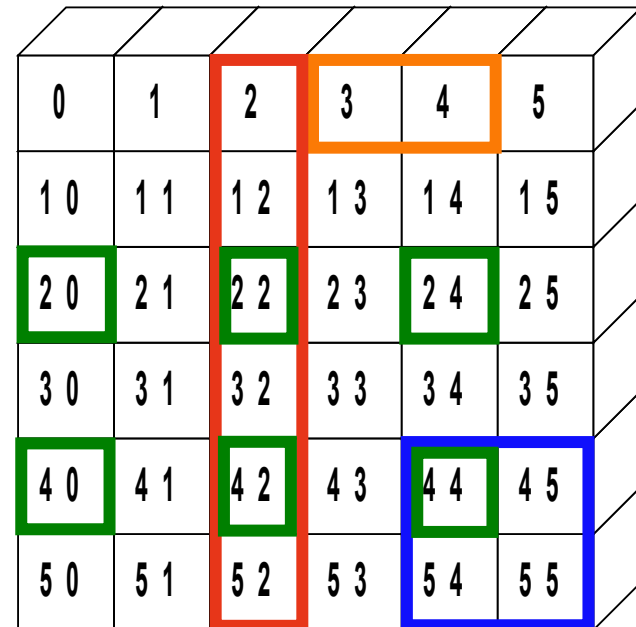
```
>>> a[0,3:5]
array([3, 4])
```

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

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

STRIDES ARE ALSO POSSIBLE

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



0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

The diagram shows a 6x6 grid of cells. The columns are indexed 0 to 5, and the rows are indexed 0 to 5. The following cells are highlighted with colored boxes:

- Column 2 (cells 20, 22, 32, 42, 52) is highlighted with a red border.
- Column 3 (cells 30, 32, 42, 52) is highlighted with an orange border.
- Column 4 (cells 40, 42, 52) is highlighted with a green border.
- Column 5 (cells 50, 52, 54, 55) is highlighted with a blue border.
- Row 2 (cells 20, 21, 22, 23, 24, 25) is highlighted with a green border.
- Row 4 (cells 40, 41, 42, 43, 44, 45) is highlighted with a green border.
- Row 5 (cells 50, 51, 52, 53, 54, 55) is highlighted with a blue border.

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)`

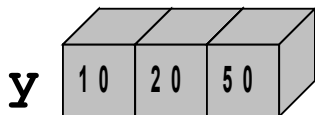
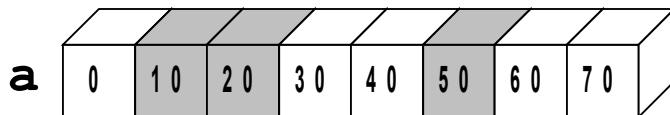
Array Functions – take ()

take (a, indices, axis=0)

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

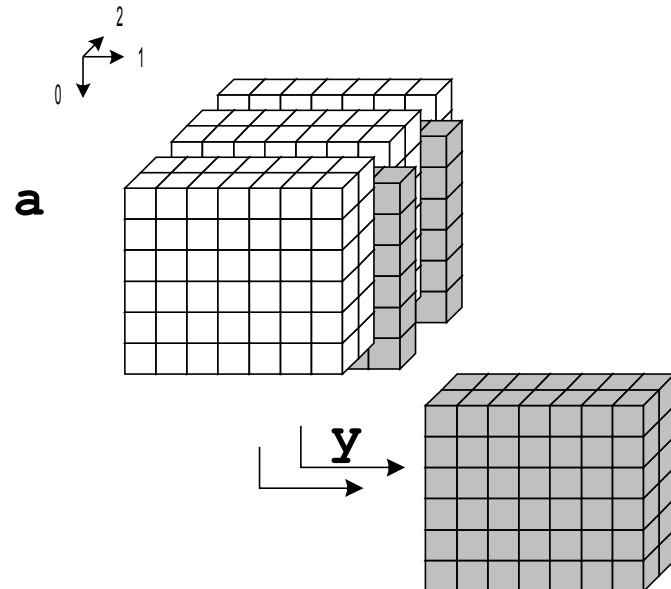
ONE DIMENSIONAL

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



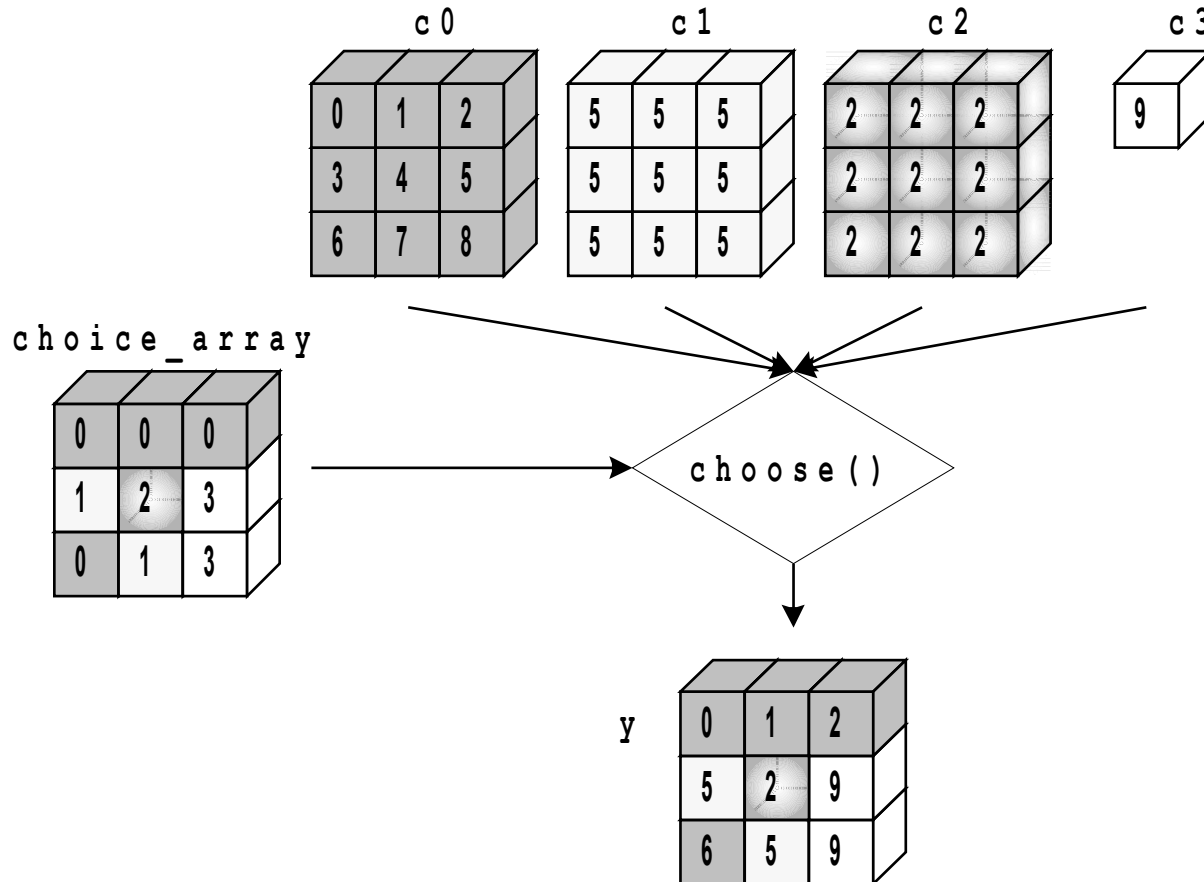
MULTIDIMENSIONAL

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



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]])
```

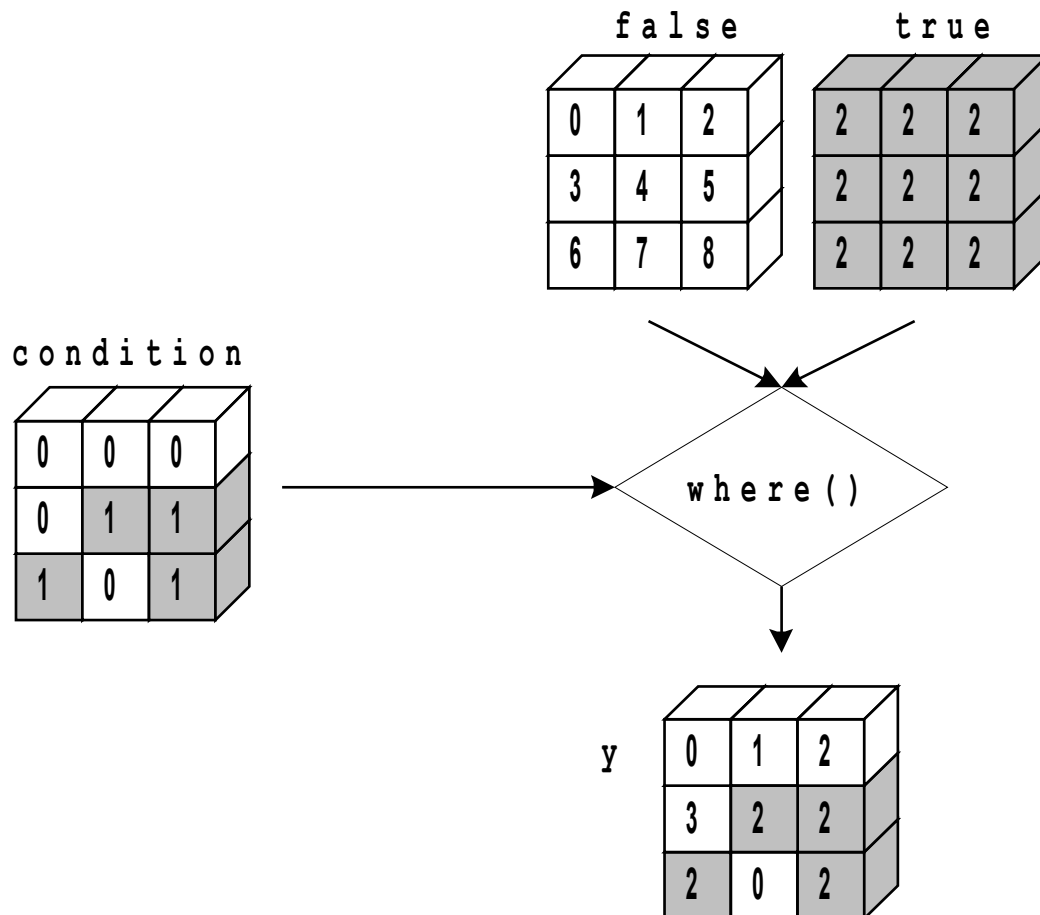
CLIP LOWER AND UPPER VALUES

```
>>> 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)
```

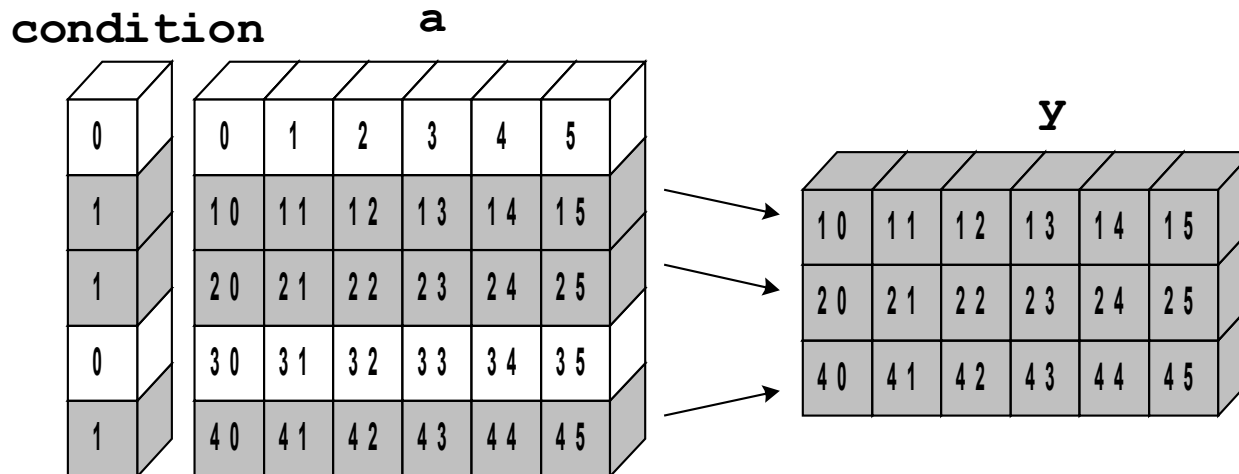


Array Functions – `compress()`

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

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

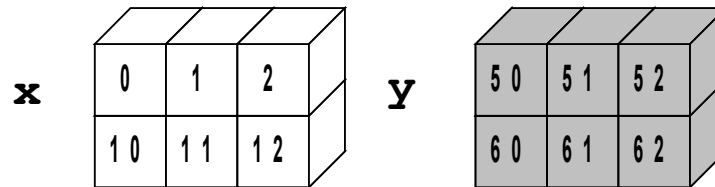
```
>>> compress(condition, a, 0)
```



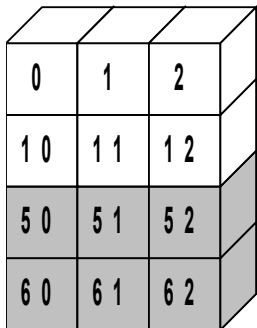
Array Functions – concatenate ()

concatenate ((a0, a1, ..., aN) , axis=0)

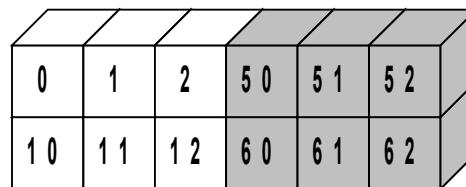
The input arrays (a0, a1, ..., aN) will be concatenated along the given axis. They must have the same shape along every axis *except* the one given.



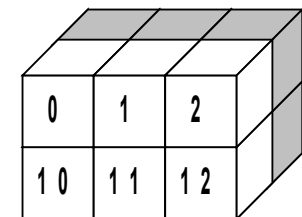
>>> concatenate ((x,y))



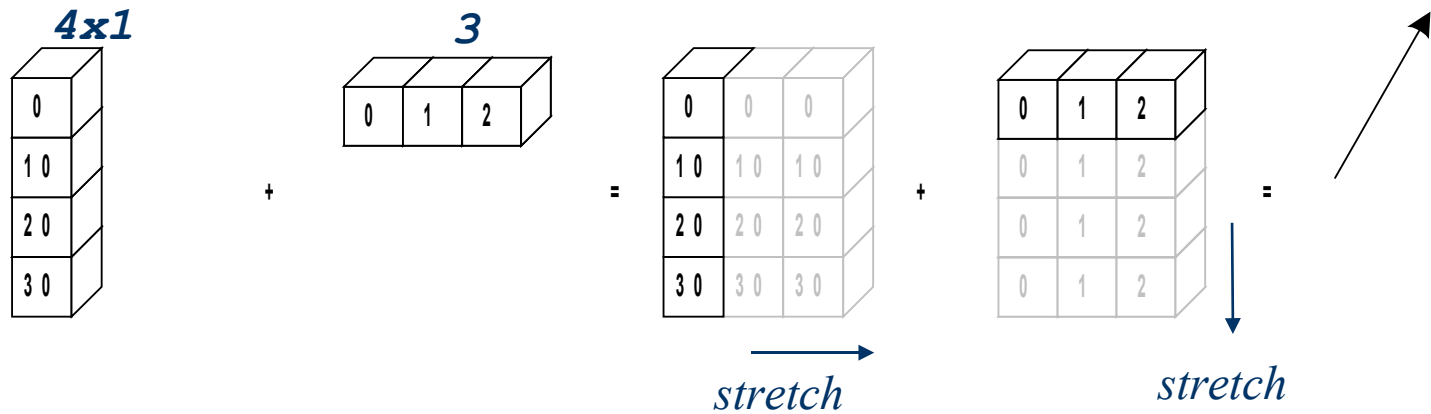
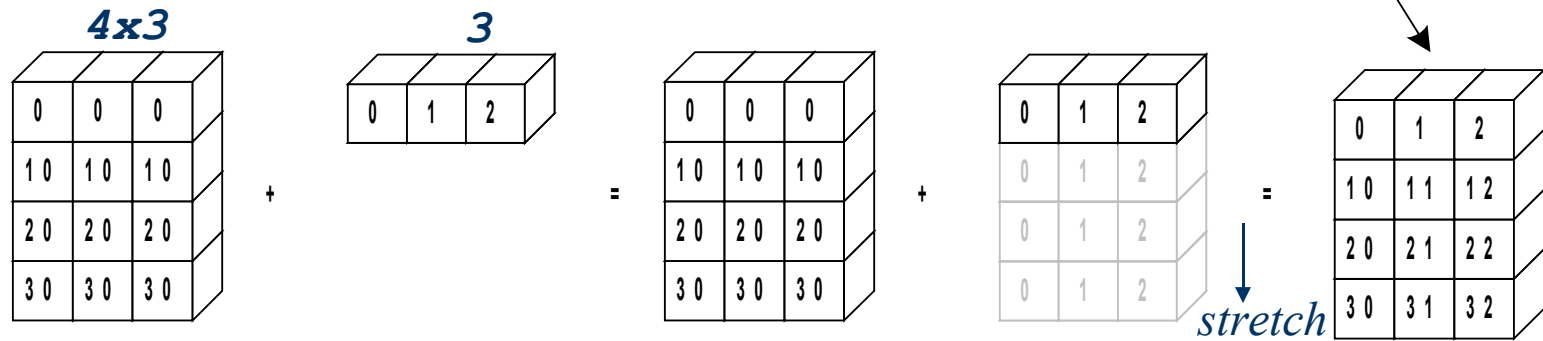
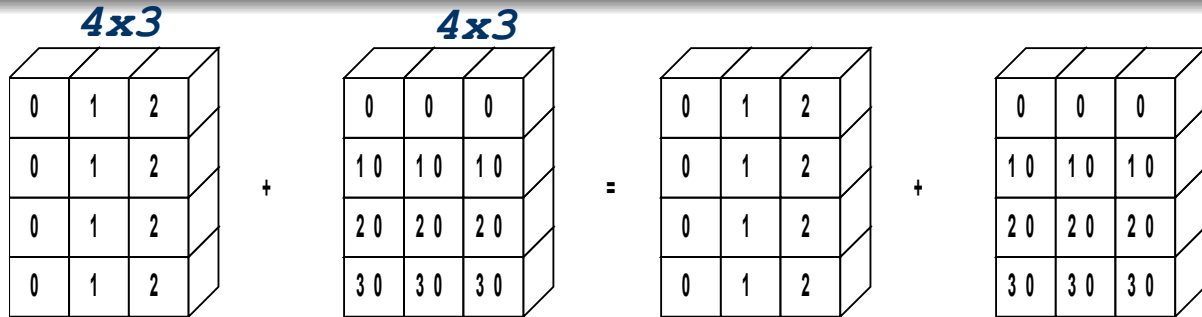
>>> concatenate ((x,y) , 1)



>>> array ((x,y))

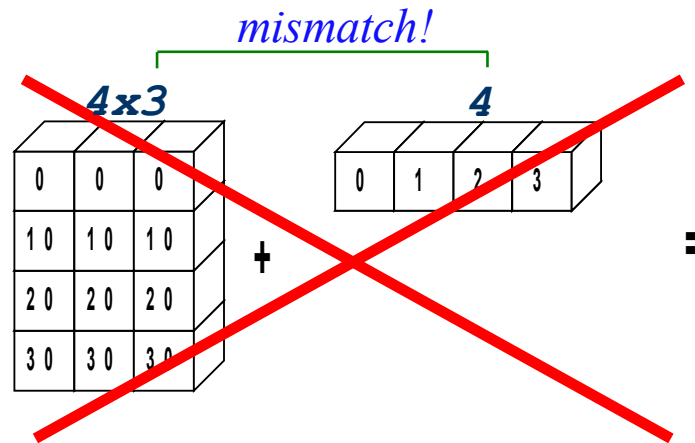


Array Broadcasting



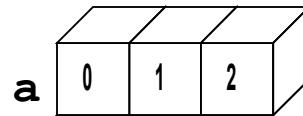
Broadcasting Rules

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



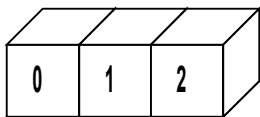
NewAxis

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



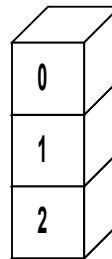
1 X 3

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



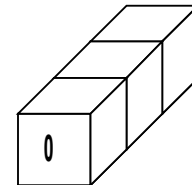
3 X 1

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



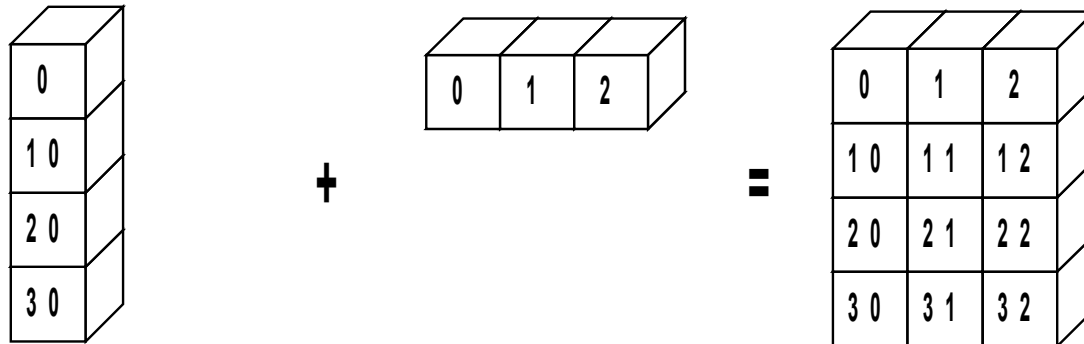
3 X 1 X 1

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



NewAxis in Action

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



Pickling

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

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



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

Performance Issues

- Interpreted, dynamic: Performance hit!
- Tim Hochberg's suggestion list:
 0. Think about your algorithm.
 1. Vectorize your inner loop:

DO NOT DO THIS:

```
z = zeros(10)
for i in xrange(10):
    z[i] = x[i] * y[i]
```

DO THIS:

```
z = x * y
```

2. Eliminate temporaries.
3. Ask for help.
4. Recode in Fortran/C/Pyrex/weave/...
5. Accept that your code will never be fast.

Step 0 should probably be repeated after every step!

SciPy

Overview

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

CURRENT PACKAGES

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

Input and Output

scipy.io --- Reading and writing ASCII files

textfile.txt

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

Read from column 1 to the end
 Read from line 3 to the end

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

```
>>> print a
```

```
[[ 98.3  94.2  95.3  91.3]
 [ 47.2  49.1  54.2  34.7]
 [ 84.2  85.3  94.1  76.4]]
```

```
>>> b = io.read_array('textfile.txt', columns=(1, -2), lines=(3, -2))
```

```
>>> print b
```

```
[[ 98.3  95.3]
 [ 84.2  94.1]]
```

Read from column 1 to the end every second column
 Read from line 3 to the end every second line

Input and Output

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

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

Class for reading and writing binary files into Numeric arrays.

Methods

- **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.

- 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
```