Scientific Python Workshop

Pat Miller

Center for Applied Scientific Computing

Feb 18-19, 2003

A drama in four acts…

- Prologue: Getting setup on Training computers
- ACT I: Basic Scientific Python
- ACT II: Using Python Numeric
- ACT III: Building Simple Extensions in Python
- ACT IV: MPI Parallel Programming in Python
- Epilogue: Resources for further study

ACT I: Basic Scientific Python

- Why Python?
- Simple Python tricks
- Running programs from Python
- Intermezzo: Pretend Physics
- Data munging
- Gnuplot: A Basic Plot package
- TKinter: Build basic GUI's
- Building and installing a package
- Steering: Leave the driving to us!

Why Python?

What is Python?

python, (*Gr. Myth.* An enormous serpent that lurked in the cave of Mount Parnassus and was slain by Apollo) **1.** any of a genus of large, non-poisonous snakes of Asia, Africa and Australia that suffocate their prey to death. **2.** popularly, any large snake that crushes its prey. **3.** totally awesome, bitchin' language that will someday crush the $'s out of certain *other* so-called VHLL's ;-) Python is an *interpreted, interactive, object-oriented* programming language. It is often compared to Tcl, Perl, Scheme or Java.

Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.

Guido van Rossum: http://www.python.org/doc/Summary.html

Why Python here at LLNL?

- Some big users
  - Python based Frameworks
    - ADiv's KULL code
    - Climate Modeling
    - Physics EOSView
  - Use Python to "steer" computation
  - Empower end-users
- A lot of little users
  - Throw away tools
  - Shareable full power tools

Simple Python tricks

- Running from the Python prompt
  - types and operators
- Using Import
- Simple functions
- If-then-else
- Lists, loops, and loop comprehensions
- Writing scripts

Running from the Python prompt

- types: int, float, complex, string, [list], (tuple)
- arithmetic: + - * / % ** abs divmod min max pow round
- convert: bool chr int long str float oct ord
- info: dir help id type
- misc: len open range/xrange zip

- TASK: Try some calculator operations at the prompt

Using Import

- >>> import math
  - may load capability (once)
  - creates a “module” object named math
- >>> from math import sin,cos,pi
  - load selected values from math
  - create local names that reference originals
- >>> from math import *
  - sledgehammer approach to load all names and will happily swat any previous or builtin names“special” names are not imported (privacy)
- >>> from math import __doc__
- >>> math.__doc__
- TASK: Are the units of sin/cos in degrees or radians?

Playing with import

- >>> import math
- >>> help(math) # Get a “man” page
- >>> help(math.sin)
- >>> dir(math) # peek at what is inside
- >>> math.pi = 16./5. # Indiana HB 246, 1897[1]
- >>> print math.pi # Don’t try this at home!

- TASK: What are the angles in a 3-4-5 right triangle?
- QUESTION: What is the difference between builtin pow() and math.pow()?

[1] The Straight Dope: http://www.straightdope.com/classics/a3_341.html

Simple functions

- One liners (statement functions)
  - >>> def f(x): return 2*x
  - >>> f(10); f(12.3); f('hello'); f([1,2,3])
  - remember the return!
- Functions see "global" values in the containing scope where they are defined
  - >>> h = 6.62606876e-34
  - >>> def hx(x): return h*x
  - >>> hx(3)
- TASK: Write a one line function named arithmetic_sum to compute Σi using the closed form n*(n+1)/2

More functions

- You can have multiple arguments
  - >>> def distance(x,y): return sqrt(x**2+y**2)
- You may define local variables, use global variables, and update global variables (with global statement)
  - >>> def f(x):
  - … x2 = x**2
  - … return x*x2
  - >>> h = 6.62606876e-34
  - >>> def g(x):
  - … h = 7 # masks the global value of h
  - … return h*x
  - >>> def z(x):
  - … global h
  - … h = 8 # Changes global value!
  - … return h*x

Still more functions

- “void” functions return None (default return!)
  - >>> def printit():
  - … print ‘hi mom’
  - … return
  - >>> x = printit()
  - >>> print x
- functions can return multiple values (sort’a)
  - >>> def quadratic_formula(a,b,c):
  - … d = (b*b-4*a*c)**.5 # not sqrt function!
  - … return (-b + d)/(2*a), (-b - d)/(2*a)
  - >>> r0,r1 = quadratic_formula(1,-1,-1)
  - >>> roots = quadratic_formula(1,-1,-1)

Functions with documentation

```
def arithmetic_sum(n):
    """ arithmetic_sum

    Compute sum of 0+1+2+…+n
    """
    return n*(n+1)/2

>>> help(arithmetic_sum)
>>> print arithmetic_sum.__doc__
```

If-then-else

- >>> if x > 10:
- … print “big”
- … elif x > 5:
- … print “medium”
- … else:
- … print “small”
- Each type has a “=0” test (often length)
- and/or are NOT boolean operations
- and/or are “short circuit” (like && || in C)

Lists and list comprehensions

- >>> [1,2,3]
- >>> [1,”hello”,None,3j]
- >>> range(10)
- >>> range(1,10)
- >>> range(1,10,2)
- >>> range(10,0,-2)
- >>> [x**2 for x in range(10)]
- >>> [x**2 for x in range(10) if x%2 == 0]

Loops

- for <values> in <driver>:
- body
- break/continue as in C
- >>> a = [1,2,3]
- >>> b = [4,5,6]
- >>> c = [[1,2],[3,4],[5,6]]
- >>> for x in range(10): …
- >>> for x in a: …
- >>> for x,y in c: …
- >>> for x,y in zip(a,b): …
- >>> for c in “hi mom!”: …

Writing scripts

- Scripts (.py files) are little Python programs
- Scripts are “byte-compiled” in 100ths of seconds into .pyc (“pic” or pee-why-see) files
- Help you reuse helper functions
- Structure tends to be…
  - DOC STRING
  - DEFINITIONS
  - if __name__ == “__main__”:
    - …
- TASK: Write a script helper.py with the functions quadratic_formula and arithmetic_sum from before

Shell like services

- >>> from glob import glob
- >>> glob(‘*.py’) # like /bin/ls
- >>> import os
- >>> os.getcwd() # like pwd
- >>> os.chdir(‘/usr/local’) # like cd
- >>> sys.argv is like argc,argv (or $1 $2 or $argv in scripts)
- string methods and re for sed/awk/tr
- open(‘foo’).read() # is like cat

Running programs from Python

- While a lot of standard services are available in Python, sometimes you just want to run an existing program
- >>> import os
- >>> os.system('ls') # None is OK, else integer status
- >>> pipe = os.popen('ls') # like a file in Python
- >>> pipe.read()
- >>> os.popen('uptime').read() # like `uptime` in shell or perl
- >>> wpipe = os.popen('sort','w')
- >>> wpipe.write('hello\nworld\nfoobar\n')
- >>> wpipe.close()

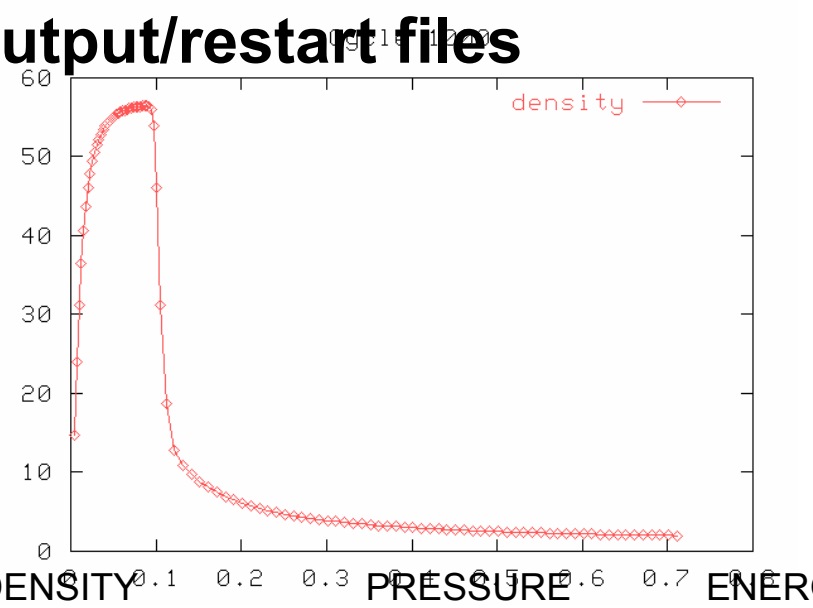
popen2()

- There is also a popen2() for bidirectional work, but buffering makes it more complicated
- >>> w,r = os.popen2('sort')
- >>> w.write('hello\nworld\nfoo\n')
- >>> w.close()
- >>> r.read()

- CAUTION: The pipes can block if the buffer overflows or if, say, the input side of the pipe (w) isn't flushed.

Intermezzo: Pretend Physics

- 1D Lagrangian hydro (spherical, E, rho, P)
- Simple data format for output/restart files




```
#MODE: 0
#CYCLE: 100
#DUMP: 10
#DT: 1.000000e-05
#GAMMA: 1.666667e+00
#NODES: 11
#ZONES: 10
# POSITION          VELOCITY          DENSITY           PRESSURE          ENERGY
0.000000e+00        0.000000e+00
1.000000e-01        -1.000000e+00     1.000000e+00      6.666667e-11      1.000000e-10
2.000000e-01        -1.000000e+00     1.000000e+00      6.666667e-11      1.000000e-10
```

- Typically, we evolve the problem forward in time and then plot field data like density vs position

Data munging

- Let's get some data
  - % lagrangian –velocity=-1 –steps=1000
  - This creates dump01000
- TASK: What's the density field?

- Approach: Write a get_density.py script
- Step 1: Get the file name (sys.argv[1])
- Step 2: open the file
- Step 3: Read line at a time until you get to the data
- Step 4: split each line (line.split()) and take the[0]'th and [2]'th item (position and density) and append to an array
- Step 5: return position and density array

re – Regular expressions

- Another way to search data and pull out items that follow set patterns
- We'll only look at a simple subset from the prompt here
- >>> import re
- >>> pattern = re.compile(r'MODE:\s\S+')
- >>> match = pattern.match(data)
- >>> match.group()
- >>> pattern = re.compile(r'MODE:\s(\S+)')
- >>> match = pattern.search(data)
- >>> match.group()
- >>> match.group(1)
- >>> match.begin()
- >>> match.end()

TASK: Rewrite the get_pressure() function to use regular expressions

Gnuplot: A Basic Plot package

- There are a variety of plotting packages for Python and some exciting new ones in the works (CHACO from www.scipy.org for the Space Telescope people)
- Also are links to high end graphics like VTK
- We'll look at a boring one ☹, but it is easy to install on Unix and gives us a lot of what we need for simple plots
- Gnuplot.py is actually implemented as a pipe on top of the gnuplot program
- You open a plotting "object" that you feed data "objects" to plot as well as plotting modifiers

Plotting

- >>> from Gnuplot import Gnuplot, Data
- >>> gp = Gnuplot(persist=1) # the “plot” object
- >>> gp.plot([(0,0),(1,1),(2,4),(3,9)])
- >>> x = range(100)
- >>> y = [xi**2 for xi in x]
- >>> gp.plot(Data(x,y))
- >>> gp.plot(Data(x,y,title=“x squared”,with=‘linespoints’))
- >>> You can overlay plots with multiple data items or “replot”
- >>> gp.hardcopy(filename=‘square.ps’)
- Try: help(gp.hardcopy)

Plotting exercises…

- >>> from Numeric import arange
- >>> from math import pi
- >>> x = 2*pi*arange(101)/100

- TASK: Plot sin(x) and sin(2x)
  - Try it with title= and with= options
  - What happens if you plot Data(x,x)?
  - Try again after
    - >>> gp('set yrange [-1:1]')

Plotting Density field

- Let's combine our ability to extract the density field with our ability to plot!
- >>> from get_density import density
- >>> x,y = density('dump01000')
- >>> gp.plot(Data(x,y,title='density',with='linespoints')

- TASK: Write a script to plot density at step 500,1000,1500
  - % lagrangian --velocity=-1 --steps=100
  - % lagrangian --file=dump00100

TKinter: Build basic GUI's

- GUIs are a great way to do data entry and value checking and even simple plotting.
- Python supports a variety of interfaces to GUI packages
- The two big ones are wxwindows and Tk
- Tk is the graphics part of Tcl/Tk (Tickle Tee Kay)
- Tkinter is Python support module

The Event Loop

- **Tkinter uses an "event-loop" which seizes control of the main thread and holds it, so it really has to run in a script.**
- **We're only going to look at a basic "dialog" pattern, but the variety in Tk is boundless**
- **We'll build buttons, labels, and data entry items….**

```
from Tkinter import *
class dialog:
    ….
if __name__ == '__main__':
   root = Tk()
   app = dialog(root)
   root.MainLoop()
```

Buttons

```
class dialog:
  def hello(self):
     print 'hi Mom!'
     return
  def __init__(self,parent):
     self.body=Frame(parent)
     self.body.pack(padx=2)  # More on this later!
     Button(self.body,text='Hello',command=self.hello).grid(row=0,column=0)
     Button(self.body,text='Quit',command=self.body.quit).grid(row=0,column=1)
     return
```

Labels

```
class dialog:
  def hello(self):
     print ‘hi Mom!’
     self.hello.set(‘good-bye’)
     return
  def __init__(self,parent):
     self.body=Frame(parent)
     self.body.pack(padx=2)  # More on this later!
     Button(self.body,text=‘Hello’,command=self.hello).grid(row=0,column=0)
     Button(self.body,text=‘Quit’,command=self.body.quit).grid(row=0,column=1)
     Label(self.body,text=‘Hello).grid(row=1,column=1,columnspan=2)
     self.hello = StringVar()
     self.hello.set(‘hello’)
     Label(self.body,textvariable=self.hello).grid(row=2,column=1)
     return
```

Data Entry

```
class dialog:
  def hello(self):
     print 'hi Mom!',self.entry.get()
     return
  def __init__(self,parent):
     self.body=Frame(parent)
     self.body.pack(padx=2)  # More on this later!
     Button(self.body,text='Hello',command=self.hello).grid(row=0,column=0)
     Button(self.body,text='Quit',command=self.body.quit).grid(row=0,column=1)
     self.entry=Entry(self.body)
     self.entry.insert(INSERT,'<default>')
     self.entry.grid(row=1,column=1)
     return
```

The big GUI

- **TASK: Build a simple GUI for Lagrangian**
  - **Should have a QUIT and PLOT button**
  - **Needs a labeled entry field for number of steps to take (default value is 0)**
  - **When you click on the PLOT button, the code should run 'lagrangian' the appropriate number of steps and then Gnuplot the density field**
  - **Extra Credit: Add a status label that indicates what's going on (waiting for input, calculating, plotting). This uses the "textvariable" option**

Building and installing a package

- Bazillions of Python modules… How do we find them?
  - http://www.vex.net/parnassus/ (Vaults of Parnassus)
  - http://starship.python.net/ (The Starship)
- Once we got it, how do we install it?
  - The miracle of “distutils” helps simplify life for the end user….

Let's build something…

- Grab a copy from the class directory...
  - % cp /usr/gapps/python/tru64_5/class/lagrangian-0.1.tgz .
  - % gunzip -c lagrangian-0.1.tgz | tar xvf -
  - % cd lagrangian-0.1
- Poke around and see what we have
  - lagrangian.c -- my original cruddy source code
  - wrapper.c -- a hand written Python interface
  - setup.py -- build control…

The Magic handshake

- % python setup.py build
- Now we have a build directory with a “temp” and “lib” directory for this version of python and the current machine configuration (this allows multiple, non-overlapping builds)

- We can try to install it….
  - % python setup.py install

- We can install it elsewhere
  - % python setup.py install --home ~
  - % python setup.py install --home ~/pub/$SYS_TYPE

Steering: Leave the driving to us!

- With steering, you “write the main()”
- Complete customization and access to low level structures allow the end user to control the package

- >>> from lagrangian import hydro
- >>> h = hydro() # zones,radius,gamma,dt,velocity,density,energy
- >>> dir(h)

Inspection

- Part of the power of steering is “inspection” -- looking at low level values...
- >>> h.velocity
- >>> h.zones
- >>> h.density

- TASK: Use Gnuplot to show initial density field

Object control

- You can override data values more easily in a scripting language than what a developer might provide...
  - >>> for j in h.zones: h.position[j] = 1.1**j
- You can mess with low level structures….
  - >>> h.velocity = h.nodes*[3]
  - >>> h.dt = 0.001
  - >>> h.step()
  - >>> h.density[3] = 2
  - >>> h.update_pressure()

Final Exam

- Using the lagrangian hydro, plot density at step 0, 500, 1000, 1500
  - >>> H = hydro(velocity=-1, zones=200)

Summary

- Python is great for general programming and short knock-off scripts
- You can run and interact with existing programs just like in bash or ksh or csh (but better!)
- You can manipulate data in text files with powerful regular expression and string functions
  - Also interfaces to data formats like PDB and HDF5
- Graphics and Plotting package interfaces
- Build complex GUIs
- Steer

More info….

- http://www.scipy.org
- http://www.python.org/doc/Intros.html
- http://starship.python.net/crew/hinsen/
- http://diveintopython.org
- http://sourceforge.net/projects/numpy
- Python Essential Reference, 2nd Edition; Beazley; ISBN: 0-7357-1091-0
- Programming Python; Lutz; ISBN: 1-56592-197-6

- Call me… I'm here to help!

A drama in four acts…

- ACT I: Basic Scientific Python
- **ACT II: Using Python Numeric**
- ACT III: Building Simple Extensions in Python
- ACT IV: MPI Parallel Programming in Python

ACT II: Using Python Numeric

- This section is mostly hands-on, at the Python prompt
- This tutorial covers similar material to David Ascher's tutorial at http://starship.python.net/~da/numtut/array.html
- But with added material and exercises

Numeric

- ***Numerical Python adds a fast, compact, multidimensional array language facility to Python.***
- **The array operators are quite FORTRAN 90ish**
- **The for element operations**
  - **infix operators are overloaded to perform the "right" thing *vector * scalar --> vector (by scalar broadcast)***
  - ***vector * vector --.> vector (by pairwise application)***
  - ***matrix * vector --> matrix (row-wise multiplication)***
  - ***matrixmultiply(matrix,vector)***
- ***Data is stored homogenously in a variety of C data types (char, sign/unsigned byte, short, int, long, float, double, complex float, complex double)***
- **Numeric.matrixmultiply, LinearAlgebra{determinate, inverse,solve_linear_equations, lapack_lite}**

Using Numeric

- >>> from Numeric import *
- >>> A = array([1,2,3])
- You can choose a type or let Numeric pick...
  - >>> array([1,2],Float32) ==> an array with 32 bit (or close) floats
  - >>> array([1,2.5]) ==> an array with Float64 data
  - >>> array([[1,2],[3,4]]) ==> a 2x2 matrix of longs

Supported types…

- char, int 8,16,32, long, float 8,16,32,64, complex 32, 64, 128, Python Object
- Some of the types may not be available
- Each array is homogenous
- Once established, an array will cling to type and shape (unlike Python arrays)
- The type (at origin) is “fitted” to the data unless otherwise specified

Why an array of Python Objects

- Hey! Isn't that just a [list]?

- DISCUSS: How are they different internally? How can that affect their use?

Concept of shape…

- Every Numeric array object has a “shape” attribute that describes its dimensionality and rectilinear shape

- >>> a = array([1,2,3,4])
- >>> a.shape

- >>> m = array([[1,2,3],[4,5,6],[7,8,9]])
- >>> m.shape

- >>> s = array(7)
- >>> a.shape

Shape is an attribute

- You can't change an item's shape, you must create a new item with a different shape (BUT WITH THE SAME DATA)
- >>> a = array([1,2,3,4,5,6])
- >>> b = reshape(a,(2,3))
- >>> c = reshape(a,(3,2))
- >>> d = reshape(b,(6,))
- >>> e = reshape(a,(2,-1))

- DISCUSSION: Why is reshape a function and not a method on the array?

Getting started with data...

- >>> ones(5) ==> array([1,1,1,1,1],Int64)
- >>> zeros(5,Float64) ==>
  array([0.0,0.0,0.0,0.0,0.0],Float64)
- >>> arange(3) ==> array([0,1,2],Int64)
- >>> arange(4,Float32) ==>
  array([0.0,1.0,2.0,3.0,4.0],Float32)
- >>> m = fromfunction(f,(10,10))
- >>> a = b.asarray('f')

The Zen of Numeric

- Very few attributes and methods
- Most operations implemented as *functions*
  - *This allows for non-Numeric arguments in a way O-O style methods would interfere with*
- The mindset is one of FORTRAN rather than one of Python wherever there are major clashes (except for the zero based indexing!)

Slicing & indexing arrays

- Work a lot like Python lists
  - BUT THEY DO NOT COPY ARRAYS!!!!!!
    - use Numeric.copy to make that happen
  - They also support multiple dimensions and strides
- >>> m = array([[1,2,3],[4,5,6],[7,8,9]])
- >>> m[0] ==> array([1,2,3])
- >>> m[:,1] ==> array([2,4,8])
- >>>m[:2,:2] ==> array([[1,2],[4,5]])

Fancy slicing….

- >>> a = reshape(arange(9),(3,3))
- >>> a[:,0]
- >>> a[::-1,0]
- >>> a[:::-1]
- >>> a[::-1,::-1]
- >>> a[…,0]

- DISCUSS: Ellipses

“Take” vs.. “Indexing”

- Indexing drops the dimensionality of a result
- taking selects within the same dimensionality

- >>> m = array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
- >>> m[0] ==> [1,2,3]
- >>>take(m,(0,)) ==> [[1,2,3]]
- >>>take(m,(0,2)) ==> [[1,2,3],[7,8,9]]
- >>>take(m,(-1,0,2)) ==> [[10,11,12],[1,2,3],[7,8,9]]

Flat, diagonal, trace

- >>> m = array([[1,2,3],[4,5,6],[7,8,9]])

- Flattening to 1D
  - m.flat ==> [1,2,3,4,5,6,7,8,9] SHARED
- Diagonal
  - >>> diagonal(m) ==> [1,5,9] copy
  - >>> diagonal(m,1) ==> [2,6] copy
  - >> >diagonal(m,-1) ==> [4,8] copy
  - Note that diagonal is a function
- Trace (sum along diagonals)
  - >>> trace(m) ==> 15 (1+5+9)

Arrays can be reshaped

- >>> a = arange(12)
- >>> reshape(a,(2,6)) ==> [[0,1,2,3,4,5],[6,7,8,9,10,11]]
- >>>reshape(a,(3,2,2)) ==>
  [[[0,1],[2,3]],[[4,5],[6,7]],[[8,9],[10,11]]]
  - 3 planes of 2 rows of 2 columns
- Reshaped arrays SHARE the same storage

Other shape modifiers…

- NewAxis
- >>> b = array([1,2,3])
- >>> b[:,NewAxis]
- >>> reshape(b,(3,1))

Filling arrays with data

- Broadcasting a scalar
  - >>> ones(5)*1.5 ==> [1.5,1.5,1.5,1.5,1.5,1.5]
- Use basic Python containers
  - >>> array(range(10))
  - >>> array(xrange(1,20,3))
  - >>> array((1,2,3))
- Use the Python “map” function
  - >>>def f(x): return x*x+3
  - >>> array(map(f, arange(0,10)))
- Use Numeric aware functions (ufunc)
  - >>> sin(arange(101)*pi/100)

UFUNC

- **Special, array aware functions that replace builtins and have other properties….**

- **>>> total = add.reduce(a)**
- **>>> rtotal = add.accumulate(a)**
- **DISCUSS: Why is this faster?**

- add, subtract, multiply, divide, remainder, power, arccos, arccosh, arcsin, arcsinh, arctan, arctanh, cos, cosh, exp, log, log10, sin, sinh, sqrt, tan, tanh, maximum, minimum, conjugate, equal, not_equal, greater, greater_equal, less, less_equal, logical_and, logical_or, logical_xor, logical_not, boolean_and, boolean_or, boolean_xor, boolean_not

More functions

- transpose(a) -- flip axes
- repeat(a,repeat)
- choose(selector, filler)
- concatenate((a0, a1, a2, …))
- ravel(a) (same as .flat)
- nonzero(a) (filter)
- where(condition, x, y)
- compress(condition,a)
- clip(m, min,max)
- dot(m1,m2)
- matrixmultiply(m1,m2)

Useful methods

- itemsize() – size of element
- iscontiguous() – stored contiguously
- typecode – element type
- byteswapped() – Change endianism
- tostring() -- serialation
- tolist() – to python lists

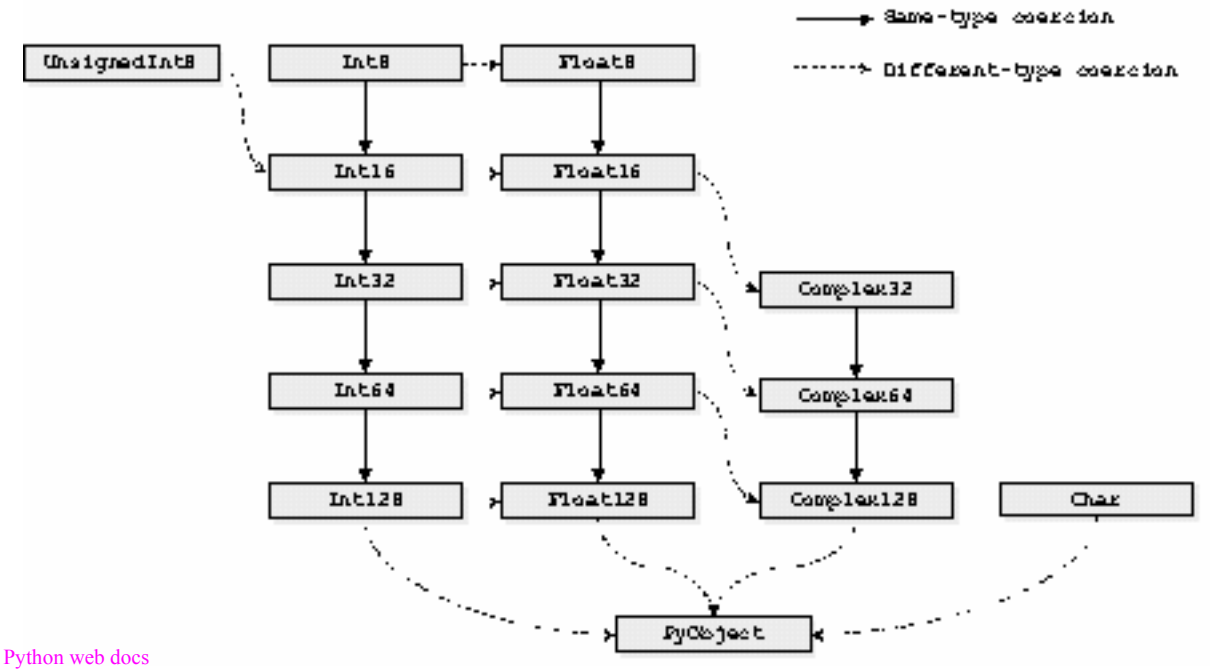
Element-wise operations

- +,-,*,/,** with normal precedence
  - >>> a = array([1,2,3,4],Float64)
  - >>> b = array([10,20,30,40],Float64)
  - >>> m = array([[11,22,33,44],[55,66,77,88]])
  - >>> a + b ==> [11,22,33,44]
  - >>> a**2 ==> [1,4,9,16]
- Where operands are not of equivalent rank, a replication takes place
  - >>> a + 3 ==>[4,5,6,7] the rank 0 scalar 3 works like a rank 1 vector
  - >>>m + b ==> [[21.,42.,63.,84.],[65.,86.,107.,128.]] (replicate the row, note the type of the final matrix)
  - >>>m * 2

Cast and coercion

Numeric performs proper coercion of its types during operations and may be forced to upcast with *asarray* function or downcast with *astype* method.

The exception is Float32 * Python float



from numerical Python web docs

Element-wise functions and reductions

- Some functions in Numeric (not math!) work element-wise across an array
  - sin, cos, abs, sqrt, log, log10, maximum, minimum
  - For functions of multiple arguments, rank replication makes arguments the same size before applying the operator,
    - >>> maximum(a, 3)
- Some reduce the elements of the array
  - sum, product,
- They can be combined and can include non-Numeric array elements like lists and tuples
  - >>> x0 = (1,1,1); x1 = (3,3,3)
  - >>> sqrt(sum((x1-x0)**2))

See *Why Isn't There a Nobel Prize in Mathematics?* by Peter Ross of UC Santa Cruz

Thinking with Vectors....

- We know that the closed form of $\sum_{n}^{i=1} i = n*(n+1)/2$. was found by a young Gauss when being punished by a tyrannical schoolmaster who assigned him the task of adding 81297+81495+....+100899 to which Gauss snapped back 9109800. He did it with clever vectors!!! Consider a smaller, simpler form with uniform stride of only 1
- 1 + 2 + 3 + ... + 9 which Gauss saw as
  - 1 + 2 + ... + 8 + 9
  - 9 + 8 + 7 + ... + 1
  - or 9 pairs that each add to 10 or sum = 9*10 / 2
- In Numeric
  - >>> a = arange(10)
  - >>> a + a[::-1]
  - >>> sum(a + a[::-1]) / 2

Exercise Gauss

- Write a Python script that does the work of young Gauss, that is find the sum of the numbers from 81297 to 100899 with uniform stride 198 by using vectors
  - answer should be an integer

Exercise Points

- You have a series of points in 3 space... Figure out the distance between adjacent pairs
  - Input: array([(0,0,0), (0,1,0), (0,1,1), (1,0,0), (1,1,0), (1,1,1),(0,0,1),(1,1,0)],Float64)
  - answer should be a vector of double precision values

Exercise Triadiagonal

- Consider a nxn tridiagonal matrix stored as d_u={du_0, du_1,...,du_{n-2}}, d={d_0,d_1,...d_{n-1}},dl={dl_0,dl_1,...dl_{n-2}}
- Write a function triinflate(du,d,dl) that returns a fully inflated matrix with all the zeros
- Write matrix multiply for a tridiagonal (du,d,dl) times a rank 1 vector v
- Write a tridiagonal system solver (Ax=b) where A is tridiagonal and b is a rank 1 vector. Use Thomas's algorithm (we'll discuss it) and don't worry about pivoting or any of that stuff.
- Use the matrixmultiply in Numeric and the solve_linear_equations in LinearAlgebra to check your work
- Example:m = array([[1,2,0],[3,4,5],[0,6,7]); b = array([10,20,30])

Final Exercise:

- **Following discussion, rewrite in Numeric...**

```
def step(u,dx,dy):
  nx, ny = u.shape
  dx2, dy2 = dx**2, g.dy**2
  dnr_inv = 0.5/(dx2 + dy2)
  err = 0.0
  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)*dnr_inv
      diff = u[i,j] - tmp
      err += diff*diff
  return sqrt(err)
```

A drama in four acts…

- ACT I: Basic Scientific Python
- ACT II: Using Python Numeric
- **ACT III: Building Simple Extensions in Python**
- ACT IV: MPI Parallel Programming in Python

ACT III: Building Simple Extensions in Python

- The TRUE power in Python is the ability to customize with your own libraries and modules.
- These “compiled assets” extend Python with speed, familiar interfaces, and bitwise compatibility with existing codes and libraries

- “From here to there. From there to here. Funny things are everywhere!” -- Dr. Suess

Overview

- Wrapping simple FORTRAN routines with pyfort
  - Write a F90'ish interface spec
  - Types map directly to Numeric arrays
  - The final interface is sort'a Fortrany
- Wrapping C routines with SWIG
  - Write a interface sort of like a .h file
  - Basic types are provided
  - Other types are handled opaquely
  - Use "typemaps" to handle weird types
- Building a module from scratch
  - With great power comes great responsibility -- A. Dumbledore

Some things we won't cover…

- things are more complicated in C++ land…
  - templating
  - classes
  - methods
  - overloading
  - memory
  - c'tor/d'tor
  - and so forth…
- CXX
- boost.Python (BPL)
  - http://www.boost.org/libs/python/doc/

PYFORT

- Believe it or not, there is a lot of great old FORTRAN code out there… Really!
- It is stupid to rewrite and test it
- The big libraries have established, well known, and well thought out interfaces
  - Interface may be driven by FORTRAN needs (e.g. providing a scratch array of some dimension)
- Numeric Python types and FORTRAN types overlap nicely
  - Is it coincidence or is it FATE!

Pyfort files

- Two kinds of files…
  - .pyf describe the interfaces to functions/subroutines
  - .pfp files describe a collection of modules
- pyfort provides a Tk GUI to build the .pfp files
- .pyf files are built by hand
  - They are tantalizingly similar to header prototypes, but are not quite the same
  - Get used to some duplication… All wrapping methods suffer from it to some degree

A really simple example…

- **% tar xvf /usr/gapps/python/tru64_5/class/examples.tar**

```
function maximus(n,x)
integer n
real x(n)
real maximus
integer I
maximus = x(1)
do I=2,n
  if ( x(I) > maximus ) maximus = x(I)
end do
return
end
```

The pyf description and the pfp project…

- **% cat maximus.pyf**

```
function maximus(n,x)
    ! The way cool and super powerful maximus function
    integer n
    real x(n)
    real maximus
end function maximus
```

- **cat fortran_demo.pfp**

```
pyf('maximus.pyf',
      sources=['maximus.f'],
      freeform=1)
```

Build and test

- % pyfort -b fortran_demo
- % cp build/lib*/*.so .
- % python
- >>> from maximus import *
- >>> maximus(7,[1,2,3,5,3,2,1])
- 5.0

FORTRAN interfaces stink ;-)

- **What you need for F77 can be clunky, redundant, and repetitive in Python**
- **Pretty clearly, the first argument to maximus (while important for Fortran77) can be computed from the actual size of the array passed in the second argument**

```
function maximus(n,x)
   ! The way cool and super powerful maximus function
   integer n=size(x)
   real x(n)
   real maximus
end function maximus
```

Returning scalars

- **They work precisely the way they should which isn't the way you want**

```
subroutine quadratic(A,B,C,R1,R2)
    real  A,B,C
    real, intent(out)::R1
    real, intent(out)::R2
end
```

```
pyf('maximus.pyf',sources=['maximus.f'],freeform=1)
pyf('quadratic.pyf',sources=['quadratic.f'],freeform=1)
```

- **>>> from quadratic import quadratic**
- **>>> quadratic(1,2,1)**

Update in place….

- **This is not a recommended practice**
  - **DISCUSS: Dangers & Hassles**
  - **But, hey, it's what the interfaces look like!**
  - **Preference is to declare one array with intent(in) and then copy into an automatically allocated array declared with intent(out)**

```
subroutine absolute(n,a)
   integer n = size(a)
   real, intent(inout)::a(n)
end
```

```
subroutine absolute(n,a,b)
   integer n = size(a)
   real, intent(in)::a(n)
   real, intent(out)::b(n)
end
```

Scratch space

- **Fortran77 interfaces sometimes require the user to provide scratch space (to make up for the lack of portable dynamic allocation)**
- **pyfort allows the user to specify intent(temporary) for arrays**
- **such space is allocated on entry to the Python function and deallocate on exit**

```
subroutine justanexample(n,a,b,scratch)
    integer n = size(a)
    real, intent(in)::a(n)
    real, intent(out)::b(n)
    real, intent(temporary)::scratch(2*n+5)
end
```

SWIG

- Developed by the infamous Dave Beazley
- Creates (from a single spec) wrappers for Python, Perl, Ruby, TCL, ocaml, guile, scheme, php, and Java
- Developed to wrap C and more or less C++
- You specify a interface descriptor file (.i file) that looks embarassingly like a .h file
- Make sure /usr/gapps/python/tru64_5/opt/bin is in your path (rehash)

Hello world in 3 files…

```
% tar xvf /usr/gapps/python/tru64_5/class/swig_examples.tar
% cat hello_implementation.c
char* hello(void) { return “hello world!”; }

% cat hello.i
%module hello
extern char* hello(void);

% cat setup_hello.py
from distutils.core import setup,Extension
setup(ext_modules=[
   Extension(‘hello’,[‘hello.i’, ‘hello_implementation.c’])
])
% python setup.py install --install-lib=.
```

Pretty simple interfaces to structures

```
% cat structure.I
%module structure
%{
#include “structure.h”
%}
%include “structure.h”

% cat structure.h
typedef struct {
   int a;
   double b;
} int_double;

extern void absolute(int_double* value);
```

Opaque types

- SWIG handles types it knows nothing about
- It creates a special “opaque” type that Python can pass around, but cannot directly manipulate
- Opaque types are great for things you are too lazy to expose or whose internal features shouldn’t be messed with from Python
- You get opaque types by omission rather than by commission. Just leave the definition out of the .i file.
- E.g. Have a function declaration like:
  - extern thing* new_thing(char*);

Using typemap for special cases

- Sometimes you don't want types to be opaque…
- You know exactly the Python object you want to stand in for that type.
- For example, a FILE* in C should be a <type file> object in Python
- You fix this with a "typemap" to teach SWIG how to handle FILE*
- This requires knowing a little about the Python-C API

A Typemap for FILE*

```
%module typemap

%typemap(in) FILE * {
   $1 = PyFile_AsFile($input);
   if ( $1 == NULL ) {
      PyErr_SetString(PyExc_ValueError,"argument must be a file");
      return NULL;
   }
}

extern void print_first_line(FILE*);
```

The shadow classes

- SWIG creates a module.py file that contains classes and interfaces
- The int_double type in the previous example, for instance, was mirrored by a normal Python class in structure.py
- This methodology make the new class available as the base of an inheritance tree

Devloping a wrapper by hand…

- Python provides literally hundreds of hooks for building and controlling functions and type objects
- We can build very simple modules with a great deal of control
  - e.g. weird function argument calling patterns
- We can build highly complex types with all the low level details set properly
- And of course, we can use our knowledge to debug SWIG and PYFORT when they bomb

Hello world by hand

- Lets start with a basic, empty module
- % cp /usr/gapps/python/tru64_5/class/*foobar* .
- % python setup_foobar.py install --install-lib=.

- The foobar module has only three attributes defined
  - __file__ : The file that holds the implementation, foobar.so
  - __name__: “foobar”, useful for introspection
  - __doc__: “” default, empty doc string

Add a void function

- Step 1) Define the function
  - static PyObject* foobar_hello(PyObject* self, PyObject* args) {…}
- Step 2) Put the function in the method table
  - {“hello”,foobar_hello,METH_VARARGS,”doc string”},
- Note how we use the Py_None value (with INCREF) to represent a void return

A drama in four acts…

- ACT I: Basic Scientific Python
- ACT II: Using Python Numeric
- ACT III: Building Simple Extensions in Python
- **ACT IV: MPI Parallel Programming in Python**

ACT IV: MPI Parallel Programming in Python

Epilogue

- http://www.python.org
- http://www.swig.org
- http://diveintopython.org
- http://sourceforge.net/projects/numpy
- http://pympi.sourceforge.net
- Python Essential Reference, 2nd Edition; Beazley; ISBN: 0-7357-1091-0
- Programming Python; Lutz; ISBN: 1-56592-197-6

UCRL-PRES-153662

Work performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48