

# **Python for Scientific Computing**

**Bill McLean, UNSW**

**Last updated on March 21, 2007**

## Outline:

1. The python language
2. Array processing with numpy
3. Graphics with matplotlib
4. Running compiled code from python

## The Python Language

Originally a teaching language written by Guido van Rossum in the early 1990s. Currently version 2.5. Features:

- free, general-purpose, interpreted scripting language
- small language with large standard library
- scalable to large applications (modules, classes, packages)
- style-neutral (procedural or object-oriented)
- dynamic types, automatic memory management
- built-in container types (lists, dictionaries)
- exception handling

Simple code example (in file roots.py)

```
from math import sqrt

def roots(a, b, c):
    A = -b / ( 2.0 * a ); B = c / a # x^2 - 2Ax + B = 0
    D = A**2 - B; s = sqrt(abs(D))
    if D > 0:
        if A >= 0:
            r1 = A + s; r2 = B / r1
        else:
            r2 = A - s; r1 = B / r2
    else:
        r1 = complex(A, s); r2 = complex(A, -s)
    return r1, r2
```

In practice, provide docstring and error checking.

```
def roots(a, b, c):  
    """  
    r1, r2 = roots(a, b, c) solves  $ax^2+bx+c=0$ .  
    """  
    if not ( isinstance(a,float) and isinstance(b,float) \  
            and isinstance(c,float) ):  
        raise ValueError, 'Coefficients must be real.'  
    ...
```

## Interactive Python session

```
> python
Python 2.5 (r25:51908, Nov 27 2006, 19:28:51)
[GCC 4.1.2 20061115 (prerelease) (SUSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from roots import roots
>>> print roots.__doc__ # or help(roots)

    r1, r2 = roots(a, b, c) solves  $ax^2+bx+c=0$ .

>>> a=3.0; b=2.0; c=4.0
>>> r1, r2 = roots(a, b, c)
>>> print r1, r2
(-0.33333333333333+1.10554159679j) (-0.33333333333333-1.10554159679j)
>>> for x in [r1, r2]: print a*x**2 + b*x + c
...
(4.4408920985e-16+0j)
(4.4408920985e-16+0j)
```

Standard mathematical functions available from libc.

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs',
'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf',
'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>> math.exp(1)
2.7182818284590451
>>> import cmath # complex version
>>> cmath.log(-1)
3.1415926535897931j
>>> math.log(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

## Array Processing with numpy

History:

- Numeric package written in 1995 by Jim Hugunin (MIT) and others; now frozen at version 24.2.
- numarray package developed by Perry Greenfield, Todd Miller and Rich White (Space Science Telescope Institute).
- scipy project begun by Travis Oliphant, Pearu Peterson, Eric Jones and others in 2001 (user can choose Numeric or numarray)
- Early in 2005, work begun on numpy, a replacement for Numeric and numarray. Stable release, numpy 1.0, in October 2006.



Indexing conventions for built-in Python list type:

```
>>> a=[1,2,3,4,'five',[6,7,8],9] # a heterogenous list
>>> len(a) # number of items in list
7
>>> a[0] # first item
1
>>> a[0:5:2] # start:(finish+1):stride
[1, 3, 'five']
>>> a[-1] # final entry
9
>>> a[-5] # same as a[-5%len(a)]
3
>>> a[7]
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: list index out of range

The `numpy` package provides a multidimensional array type and the facilities to achieve high performance with vectorized operations.

```
>>> from numpy import *
>>> a = arange(12, dtype='float32')
>>> print a
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
>>> a.resize((3,4))
>>> print a                                # row-major order by default
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
>>> a.itemsize
4
>>> print a.shape, a.nbytes # each entry takes 4 bytes
(3, 4) 48
```

```
>>> print a.T          # transpose
[[ 0.  4.  8.]
 [ 1.  5.  9.]
 [ 2.  6. 10.]
 [ 3.  7. 11.]]

>>> b=a[:,::-1]       # reverse order of cols

>>> print b
[[ 3.  2.  1.  0.]
 [ 7.  6.  5.  4.]
 [11. 10.  9.  8.]]

>>> print a*b         # elementwise array arithmetic
[[ 0.  2.  2.  0.]
 [28. 30. 30. 28.]
 [88. 90. 90. 88.]]
```

```
>>> a[0,:]=-1
>>> print b                                # a and b reference the same data
[[ -1.  -1.  -1.  -1.]
 [  7.   6.   5.   4.]
 [ 11.  10.   9.   8.]]
>>> c=a.copy()
>>> c[:,2]=-2
>>> print c
[[ -1.  -1.  -2.  -1.]
 [  4.   5.  -2.   7.]
 [  8.   9.  -2.  11.]]
>>> print a                                # a and c reference different data
[[ -1.  -1.  -1.  -1.]
 [  4.   5.   6.   7.]
 [  8.   9.  10.  11.]]
```

```
>>> A=matrix(a); B=matrix(b) # interpret arrays as matrices
>>> print A * B.T           # matrix multiplication
[[  4.  -22.  -38.]
 [ -22.  116.  204.]
 [ -38.  204.  356.]]
>>> A=mat('3.,0,-1;4,5,0;7,-1,3') # matlab syntax
>>> print A
[[ 3.  0. -1.]
 [ 4.  5.  0.]
 [ 7. -1.  3.]]
>>> b = mat('-1; 14; 17.')
```

```
>>> print b
[[ -1.]
 [ 14.]
 [ 17.]]
```

```
>>> x = linalg.solve(A, b)    # solve Ax=b
>>> r=b-A*x                  # residual
>>> print r
[[ 0.]
 [ 0.]
 [ 0.]]
>>> a=zeros((3,6),order='fortran')
>>> for n in range(a.shape[1]):
...     a[:,n]=arange(n*a.shape[0]+1,(n+1)*a.shape[0]+1)
...
>>> print a
[[ 1.   4.   7.  10.  13.  16.]
 [ 2.   5.   8.  11.  14.  17.]
 [ 3.   6.   9.  12.  15.  18.]]
```

```
>>> print a.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> isfortran(a)
True
>>> b=fromfunction(lambda i,j:i-j,(3,5))
>>> print b
[[ 0. -1. -2. -3. -4.]
 [ 1.  0. -1. -2. -3.]
 [ 2.  1.  0. -1. -2.]]
>>> c=b.T
>>> print isfortran(b), isfortran(c)
False True
```

## Graphics with matplotlib

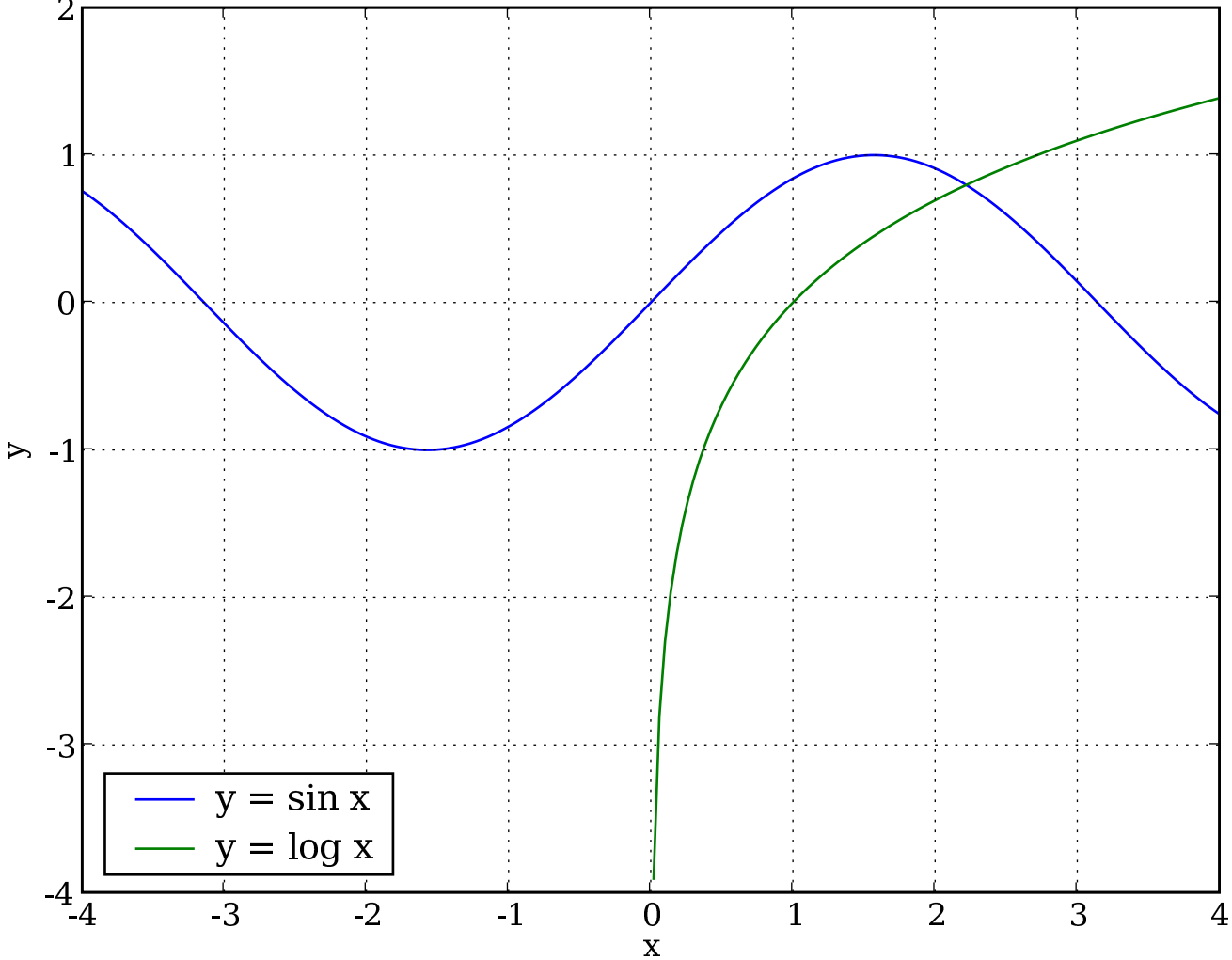
- Principal developer is John Hunter.
- High-quality 2D graphics, some basic 3D.
- pylab module provides a matlab-like API.
- library provides an abstraction layer between python and a variety of graphics backends: GTK, GTKAgg, GTKCairo, WX, WX-Agg, FltkAgg, QtAgg, TkAgg, Agg, Cairo, GD, GDK, Paint, PS, PDF, SVG.
- popular for embedding graphics in GUI applications.
- requires numpy.



Simple example:

```
>>> from pylab import *
>>> x = linspace(-4, 4, 200)
>>> plot(x, sin(x), x, log(x))
Warning: invalid value encountered in log
>>> grid(True); xlabel('x'); ylabel('y')
>>> title('Two Plots')
>>> legend(['y = sin x', 'y = log x'], loc='lower left')
>>> savefig('myplot.eps')
```

Two Plots



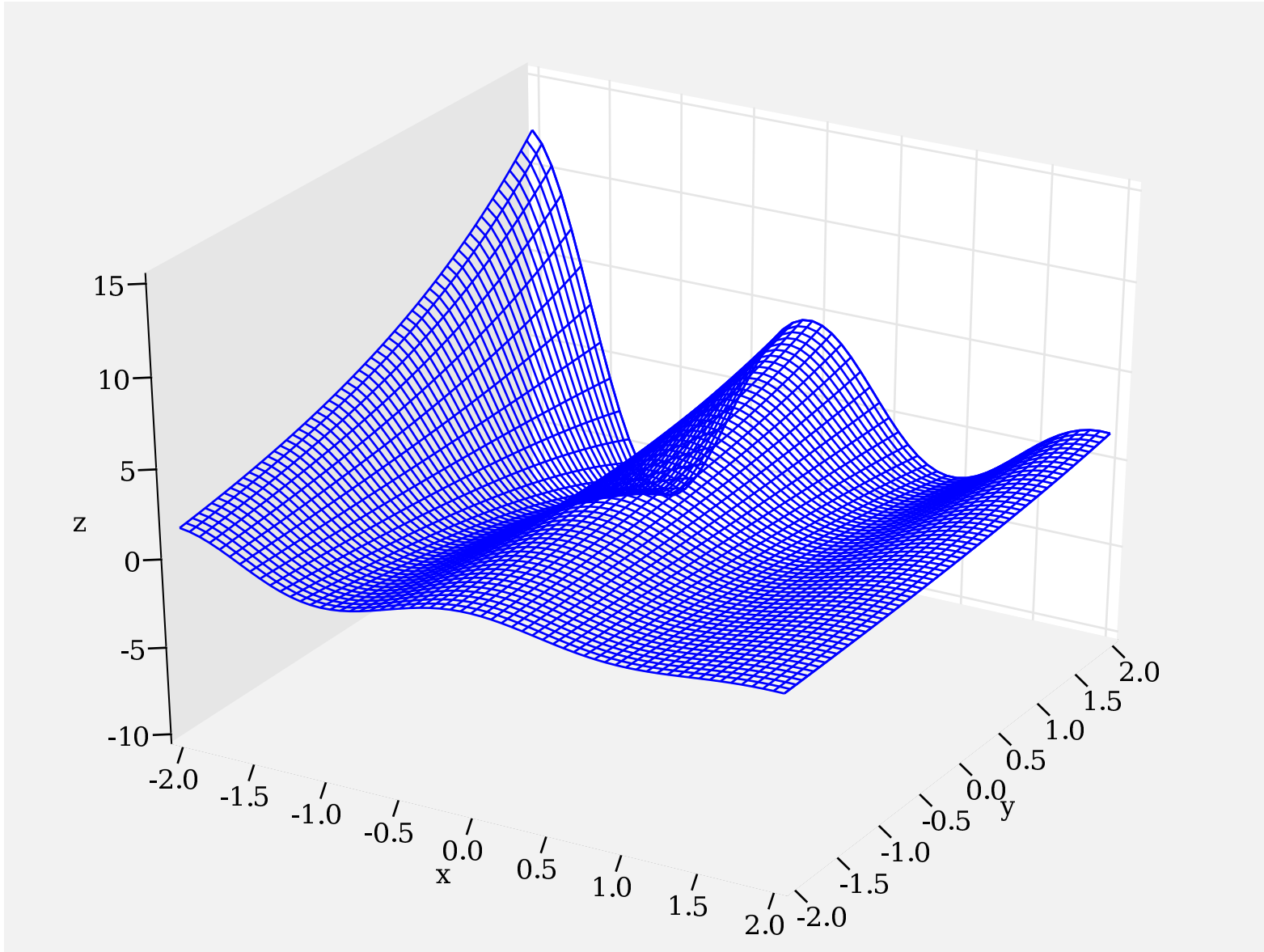
```
from numpy import *
from matplotlib.axes3d import Axes3D
from pylab import figure, show, savefig

x = y = linspace(-2, 2, 60)
[X, Y] = meshgrid(x, y)
Z = exp((Y-X+1)/2) * cos(pi*X)

fig = figure(1); ax3d = Axes3D(fig)
surf = ax3d.plot_wireframe(X, Y, Z)

ax3d.set_xlabel('x'); ax3d.set_ylabel('y')
ax3d.set_zlabel('z')

savefig('simple3d.eps')
```



## Running compiled code from python

Motivation:

- (a) speed up key parts of a python application;
- (b) make use of existing codes.

Two general approaches:

1. (low-tech) system calls, command-line args and files;
2. (hi-tech) load shared library and pass data (pointers) directly to and from compiled routines.

Tools for second approach: f2py, ctypes, swig, pyrex, weave, ...

The `numpy` package provides some basic routines for `fft`, random numbers and linear algebra; many more are available in `scipy`, including quadrature, odes, interpolation, special functions and more linear algebra.

Consider a simple example in which we want to initialise an  $m \times n$  array,

$$a_{ij} = \frac{e^{-x_i y_j} + \cos(\pi x_i)}{1 + y_j^2},$$

using the fortran code

```
do i = 1, m
  a(i,n) = cos(PI*x(i))
end do
do j = 1, n
  do i = 1, m
    a(i,j) = ( a(i,n) + exp(-x(i)*y(j)) ) / ( 1 + y(j)**2 )
  end do
end do
```

Put the loops inside a subroutine with the interface

```
subroutine matrix_init(m, n, x, y, a)
  integer,      intent(in)  :: m, n
  real(kind=8), intent(in)  :: x(m), y(n)
  real(kind=8), intent(out) :: a(m,n)
```

and put the subroutine in a source file `fsub.f95`. The command

```
> f2py -c --fcompiler=g95 -m fsub fsub.f95
```

will automatically create a python extension module `fsub.so`.

```
>>> from numpy import *
>>> import fsub
>>> help(fsub)
This module 'fsub' is auto-generated with f2py (version:2_3473).
Functions:
    a = matrix_init(x,y,m=len(x),n=len(y))
>>> x=linspace(0,1,3)
>>> y=linspace(-1,1,4)
>>> a=fsub.matrix_init(x, y)
>>> print a
[[ 1.          1.8          1.8          1.          ]
 [ 0.82436064  1.06322437  0.76183355  0.30326533]
 [ 0.85914091  0.35605118 -0.25512182 -0.31606028]]
```



Alternatively, we might write a C function.

```
#include<math.h>

void matrix_init(int m, int n, double x[m], double y[n],
                double a[m][n])
{
    int i, j;
    double cxi;

    for (i=0; i<m; i++) {
        cxi = cos(M_PI*x[i]);
        for (j=0; j<n; j++)
            a[i][j] = ( exp(-x[i]*y[j]) + cxi )
                    / ( 1 + pow(y[j],2) );
    }
}
```

If the source file is `csub.c` then we build a shared library `csub.so` as follows:

```
> gcc -c -fPIC csub.c
> gcc -shared -o csub.so csub.o -lm
```

The `ctypes` module allows python applications to access routines in a shared library.

```
from numpy.ctypeslib import ndpointer, load_library
from ctypes import c_int

csub = load_library('csub.so', '.')
```

The best approach is to provide a python “wrapper” function:

```
def matrix_init_c(x, y):
    if isinstance(x, ndarray) and isinstance(y, ndarray):
        m = x.size
        n = y.size
    else:
        raise ValueError, 'x and y must be numpy arrays'
    m = x.size
    n = y.size
    c_array = ndpointer(dtype='float64', flags='CONTIGUOUS')
    csub.matrix_init.argtypes = [c_int, c_int, c_array, c_array, c_array]
    a = zeros((m,n), dtype='float64')
    csub.matrix_init(m, n, x, y, a)
    return a
```

On my laptop, I found that with  $m = n = 3000$ , the fortran and C codes both took 0.43 seconds. By comparison, the numpy code

```
m = x.size;      n = y.size
x.resize((m,1)); y.resize((1,n))
a = ( exp(-x*y) + cos(pi*x) ) / ( 1 + y**2 )
```

took 0.72 seconds (or 65% longer). Using nested python loops,

```
m = x.size; n = y.size
a = zeros((m,n), dtype='float64')
for i in arange(m):
    cxi = mcos(pi*x[i])
    for j in arange(n):
        a[i,j] = ( mexp(-x[i]*y[j]) + cxi ) / ( 1 + y[j]**2 )
```

took 43 seconds (or 100 times longer).

The semi-vectorized python code,

```
for i in arange(m):  
    a[i,:] = ( exp(-x[i]*y) + mcos(pi*x[i]) ) / ( 1 + y**2 )
```

took 0.68 seconds, i.e., the same as the fully-vectorized code.

Despite their names, `f2py` can handle C codes and `ctypes` can handle fortran codes.

Further information:

[www.python.org](http://www.python.org)

[www.scipy.org](http://www.scipy.org)