Scientific Computing in Python

Konrad Hinsen
Laboratoire de Dynamique Moléculaire
Institut de Biologie Structurale – Jean-Pierre Ebel
41, av. des Martyrs
38027 Grenoble Cedex 1, France
hinsen@ibs.fr

14 October 1997

Contents

Packages and programs used in this tutorial

This tutorial uses the following packages in addition to the standard Python library:

- The Python numerics extension (aka NumPy) was written by Jim Hugunin and is available from http://www.sls.lcs.mit.edu/jjh/numpy/. It contains the modules `Numeric`, `LinearAlgebra`, and `FFT`.
- The module `real` that implements unlimited-precision real number was written by Jurjen N.E. Bos and is available from http://www.python.org/ftp/python/contrib/Math/real-accurate.pyar.
- The netCDF interface module is available from http://starship.skyport.net/crew/hinsen/netcdf.html
- A collection of scientific modules is available from http://starship.skyport.net/crew/hinsen/. It contains all the other non-standard modules used in this tutorial.

The utility programs mentioned are

- The Simplified Wrapper and Interface Generator (SWIG), written by David Beazly, is available at http://www.cs.utah.edu/˜beazley/SWIG/swig.html.

Interactive Python: the ultimate desk calculator

Interactive use is very important in scientific computing.
Recommendations:

- For Emacs users: use the special Python mode for convenient interactive work.
- The graphical user interface PTUI provides similar services without Emacs.
- Prepare an “interactive” module which imports everything you typically need (e.g. “`from Numeric import *`”) and have it loaded automatically for interactive sessions.

Scientific data types and functions

Scientists work with descriptions of real-life objects and with mathematical abstractions. Both can be represented by special data types.
Real-life objects:

- Physics: electrons, atoms, crystals, wave functions
- Chemistry: molecules, orbitals
- Biology: proteins, cells, plants, animals
- Meteorology: atmosphere
- General: experimental setups, lab equipment

Mathematical abstractions:

- Numbers: integers, quaternions, intervals
- Geometric: lines, vectors, transformations
- Algebraic: matrices, groups
- Mappings: functions, derivatives

Many common operations can be implemented as methods for such data types.
However, some operations are better expressed by a traditional procedural approach, e.g. mathematical functions.

Numbers

Built-in number types:

- Integers (`int`): `0, 1, 2, 3, -1, -2, -3`
- Long integers (`long`): `0L, 1L, 2L, 3L, -1L, -2L, -3L`
- Real numbers (`float`): `0., 1., -0.00367, -2.3e-5`
- Complex numbers (`complex`): `1j, -3.j, 2.-4.5j`

The precision of real and complex numbers is determined by the C compiler (type `double`); usually 64-bit IEEE format, i.e. 15 decimal digits. Standard integers are equivalent to the C type `long`, which usually has 32 bits or more. Long integers can have arbitrary length.

Standard arithmetic operations:

- Addition, subtraction: `a+b, x-3.`
- Multiplication, division: `x*1j, 2./3.`
- Modulus: `5%3, 3.5%2.8`
- Power: `x**2, (2.+3.j)**(4-3.5j)`

When operands of unequal type are combined, the result is of the "higher" type (integer–real–complex).

Caution: *Division of two integers returns an integer, i.e.* `1/3` *is zero!*

More number types can be provided by modules, e.g.:

- Unlimited precision real numbers in the module `real`.
- Quaternions in the module `Quaternion`.

Common mathematical functions in the module `Numeric`:
`sqrt`, `log`,`log10`, `exp`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sin`, `cosh`

Note: The standard library contains mathematical functions for real numbers in the module `math` and for complex numbers in `cmath`. The functions from `Numeric` can be applied to any number type.

Geometry

Class `Vector` in the module `Vector` implements vectors in 3D space:

```
from Vector import Vector

x = Vector(2.,3.,4.)
y = Vector(0.,0.,1.)
print x+y, x-y    # addition, subtraction
print 2*y+x/3.5   # multiplication/division by scalars
print x*y         # dot product
print x.cross(y)  # cross product
print x.length()  # length
print x.normal()  # a normal vector parallel to x
print x.angle(y)  # the angle between x and y
print x[0], x[2]  # accessing components
```

Vectors are *immutable*; the components cannot be changed.
The elements can be of any number type, e.g. also unlimited precision numbers etc.

Class `Tensor` in the module `Tensor` implements tensors in 3D space:

```
from Tensor import Tensor
# Tensor constants: Kronecker and Levi-Civita tensor
from Tensor import delta, epsilon

rank1 = Tensor([1., 2., 3.])
rank2 = Tensor([ [0., 1., 1.],
                 [2., 1., 3.],
                 [5., 1., 8.] ])
# Arithmetic
print rank1+rank1
print rank2-rank2
print 3*rank2
# Tensor product; the result has rank 2
print rank1*rank1
# Transpose: reverse index order
print rank2.transpose()
# Contraction over last index of first
# and first index of second tensor
print rank2.dot(rank1)
# Some operations for rank 2 tensors only:
print rank2.trace()
print rank2.symmetricalPart()
print rank2.asymmetricalPart()
print rank2.eigenvalues()
```

The module `Transformation` contains classes `Translation` and `Rotation`, which describe transformations in 3D space:

```
from Transformation import Translation, Rotation
from Vector import Vector
from Numeric import pi

# Translation by displacement vector
t = Translation(Vector(0.,0.,2.5))
# Rotation by axis and angle
r = Rotation(Vector(1.,0.,0.), pi/2)
# Composition: first r then t
combined = t*r
# inversion
inverse = combined.inverse()
# Application to a vector
x = Vector(2., 1.5, 3.)
x_transformed = combined(x)
# Extract translational part
t = combined.translation()
# Extract rotational part
r = combined.rotation()
# Get displacement vector
v = t.displacement()
# Get axis and angle
axis, angle = r.axisAndAngle()
```

Some applications:

- Rotate a point `p` (a vector object) around a line parallel to the x-axis through the point `o` (another vector object) by the angle $\pi/4$ ($45°$):

  ```
  transformation = Translation(o) * \
                   Rotation(Vector(1,0,0), pi/4.) * \
                   Translation(-o)
  p = transformation(p)
  ```

- If you rotate by $20°$ around the x-axis and then by $75°$ around the z-axis, what's the axis and angle of the total rotation?

  ```
  degrees = pi/180.
  total = Rotation(Vector(0,0,1), 75*degrees) * \
          Rotation(Vector(1,0,0), 20*degrees)
  ```

```
    axis, angle = total.axisAndAngle()
    print "Axis: ", axis
    print "Angle: ", angle/degrees
```

Arrays

Arrays are multidimensional sequence objects. Some applications:

- Linear algebra (vectors, matrices)
- Experimental data (tables)
- Values on a grid (e.g. finite element methods)

Arrays are a very powerful data type with many special operations. They are covered in detail in a separate tutorial. Only some basic operations are presented here.
Arrays and array functions are implemented in the module `Numeric`:

```
from Numeric import *

# A one-dimensional array of reals
a = array([0., 2., -1.5, -0.3])

# A two-dimensional integer array
b = array([ [ 1, -4,  0, 3],
            [-6,  2, -1, 5] ])

# Arithmetic
print a/1.5  # divide each element by 1.5
print b-3    # subtract 3 from each element
print a*b    # combine a with each row of b


# Dot product
print dot(b, a)

# Indexing
print a[0], a[2:4]     # just like lists
print a[1::2], a[::-1] # third number indicates stride
print b[0]             # only first index
print b[1, 3], b[:, 3] # multidimensional indices

# Transpose (reverse axis order)
print transpose(b)
```

The mathematical functions from the module `Numeric` can be applied directly to arrays.
Arrays can often be interchanged with nested lists. However, all elements of an array are of the same type (integer, float, complex), except for general object arrays, which are not treated here.
Array operations are very much faster than equivalent operations on lists with explicit loops!

Caution: *Subarrays extracted by indexing share their data space with the original array, i.e. changes to one of the arrays will affect the other. This is an intentional and useful feature, but it also leads to frequent errors.*

Linear Algebra

The module `LinearAlgebra` contains the most common operations in linear algebra. It is based on the LAPACK library. Vectors and matrices are represented by arrays of rank 1 and 2. Arguments can be integer, real, or complex, with integer arrays being converted to real.

Linear equations
Solution of linear equations uses LU decomposition and back substitution.

```
import Numeric;         N = Numeric
import LinearAlgebra;   LA = LinearAlgebra

a = N.array([ [1., 2.],
              [3., 4.] ])
b = N.array([7., 8.])
x = LA.solve_linear_equations(a, b)

# Several right-hand sides:
```

```
b = N.array([ [ 0., -2., -5.],
              [ 7.,  3.,  2.] ])
x = LA.solve_linear_equations(a, b)
```

Matrix inversion is handled by supplying an appropriate set of right-hand side vectors.

```
a_inv = LA.inverse(a)
```

Generalized inverses are calculated via singular value decomposition (see below).

```
b_inv = LA.generalized_inverse(b)
```

Eigenvalue problems
Eigenvalues are returned in a rank 1 array, eigenvectors in a rank 2 array with one eigenvector per row. For non-symmetric arrays, the right eigenvectors are calculated. Use the transpose to get the left eigenvectors.

```
ev = LA.eigenvalues(a)
ev, vectors = LA.eigenvectors(a)
```

Singular value decomposition returns three arrays `u`, `s` and `vt`. Array `s` has rank 1 and contains the singular values, i.e. the diagonal elements of the singular value matrix Σ. The other two arrays have rank 2 and represent the orthogonal transformation matrices. The original matrix is equal to $U \cdot \Sigma \cdot V^T$.

```
u, s, vt = LA.singular_value_decomposition(b)
```

Fourier Transforms

The module `FFT` contains Fast Fourier Transform routines, based on FFTPACK:

```
import Numeric;  N = Numeric
import FFT

x = N.array([0., 1., 2., 3., 2., 1.], N.Complex)
x_fft = FFT.fft(x)
x_test = FFT.inverse_fft(x_fft)
print x_test
```

```
x = N.array([ [0., 1., 0., 1.],
              [1., 0., 1., 0.],
              [0., 1., 0., 1.],
              [1., 0., 1., 0.] ], N.Complex)
x_fft = FFT.fft2d(x)
x_test = FFT.fft2d(x_fft)/(x.shape[0]*x.shape[1])
print x_test
```

Statistics

The module `Statistics` contains elementary statistics functions that work on any sequence object (lists, arrays, etc.):

```
import Statistics
import RandomArray
import Gnuplot

data = RandomArray.random(500)

print Statistics.average(data)
print Statistics.variance(data)
print Statistics.standardDeviation(data)
Gnuplot.plot(Statistics.histogram(data, 50))
```

Parameter fitting

The module `LeastSquares` implements the Levenberg-Marquardt algorithm for general non-linear least-squares fits.
First step: define the function to be fitted as a Python function. It can depend on any number of variables and parameters. It is called with two arguments: a tuple of parameters, and a tuple with the values of the variables. The return value must be a number.

Example: The function e^{ax+by} could be written as

```
import Numeric

def two_exponentials(parameters, values):
    a, b = parameters
    x, y = values
    return Numeric.exp(a*x+b*y)
```

The data must be prepared as a sequence (e.g. a list) of data points. Each data point is a sequence of length two or three. Its first element specifies the independent variables (x and y in the example) as a tuple. The second element is the measured value at this point, a number. The optional third element (default 1.) is the statistical variance (the inverse of the weight) of the data point.

Finally, an initial estimate for the parameters must be provided. Then the fitting procedure can be started:

```
from LeastSquares import leastSquaresFit

data = [ [(0., 1.),    20.],
         [(1., 0.),     7.],
         [(2., 1.),  1100.],
         [(0., 2.),   400.] ]

initial = (1, 1)

param, error = leastSquaresFit(two_exponentials,
                               initial, data)
print "Fitted parameters: ", param
print "Fit error: ", error
```

The fit procedure uses *automatic derivatives* to calculate the exact derivatives of the model for local linearization. See the modules `Derivatives` and `FirstDerivatives` for details.

Interpolation

Data defined on a grid can be interpolated to yield a continuous mathematical function defined at any point. The module `Interpolation` defines a class that represents such a function.

```
from Interpolation import InterpolatingFunction
import Numeric

grid = Numeric.array([0., 1., 2., 3., 4.])
values = Numeric.sqrt(grid)
f = InterpolatingFunction((grid,), values)
```

The first argument is a tuple, because functions of several variables need more than one grid axis. Interpolating functions can be defined for any number of variables, and the values can be multidimensional as well. For example, a vector field (a three-component function of three variables) would have three grid axes and a rank 4 value array with a length of three for the last dimension.
An interpolating function can be called just like any other function:

```
print f(1.), f(2.5), f(3.1415926)
```

There are also differentiation and integration operations, which return another interpolating function. The definite integral returns either a function or a number, if no independent variables remain:

```
f_deriv = f.derivative()
f_int = f.integral()
f_defint = f.definiteIntegral()
```

For functions of several variables, all these operations take an optional argument indicating the variable; the default is 0.
To obtain the definite integral on a subinterval, create an interpolating function defined on this subinterval by

```
f_subinterval = f.selectInterval(first, last)
```

Example with two variables

```
from Interpolation import InterpolatingFunction
import Numeric

grid = Numeric.array([0., 1., 2., 3., 4.])
values = Numeric.exp(Numeric.add.outer(grid, 0.5*grid))
f = InterpolatingFunction((grid, grid), values)
```

```
f_diff_y = f.derivative(1)
f_diff_xy = f_diff_y.derivative(0)

print f_diff_xy(1.5, 2.5)
```

Example of a two-component function

```
from Interpolation import InterpolatingFunction
import Numeric;  N = Numeric

grid = N.array([0., 1., 2., 3., 4.])
values = N.transpose(N.array([N.sin(0.1*grid), N.cos(0.1*grid)]))
f = InterpolatingFunction((grid,), values)

print f(3.5)
```

Polynomial fits

A polynomial of a given order can be obtained from the grid data by least-squares fitting. Polynomials are defined by class `Polynomial` in module `Polynomial`.

```
from Interpolation import InterpolatingFunction
import Numeric

grid = Numeric.array([0., 1., 2., 3., 4.])
values = Numeric.sqrt(grid)
f = InterpolatingFunction((grid,), values)
f_polynomial = f.fitPolynomial(3)

print Numeric.sqrt(3.4), f(3.4), f_polynomial(3.4)
```

Polynomials can be fitted to functions of several variables, but the values of the functions must be simple numbers.

Define your own data types

Programs become much simpler if problem-specific data types are available. Since their definition is easy in Python, it is usually a good investment.

Here is a simplified definition of class `Vector`. The real one has more methods and some optimizations.

```
import Numeric

class Vector:

    def __init__(self, x, y, z):
        self.array = Numeric.array([x,y,z])

    def __add__(self, other):
        sum = self.array+other.array
        return Vector(sum[0], sum[1], sum[2])

    def __mul__(self, other):
        return Numeric.dot(self.array, other.array)

    def length(self):
        return Numeric.sqrt(self*self)

    def __repr__(self):
        return 'Vector(%f, %f, %f)' % tuple(self.array)
```

Note: Python code tends to be readable, so study some modules of interest to you to learn about implementation strategies!

Getting data into and out of files

The basics: file I/O to and from strings
Writing to a file:

```
file = open('an_example', 'w')
file.write('This is the first line.\n')
file.write('This is the second line.\n')
file.write('This is  ')
file.write('the last line')
file.write(' of this file.\n')
file.close()
```

Reading from a file line by line:

```
file = open('an_example', 'r')
while 1:
    line = file.readline()
    if not line: break
    # print without the final newline
    print line[:-1]
file.close()
```

Note: Like C, but unlike Fortran, Python uses a stream-based I/O model. A file is a sequence of characters; the line structure is indicated by special control characters (`'\n'`). Fortran I/O is record-based: each read or write statement treats one record (which is one line for text files).

"Value added" text files
The module `TextFile` defines a class `TextFile` which adds some convenience options to bare Python files:

- transparent handling of compressed files: add the extension `'.Z'` or `'.gz'` to the file name and forget about compression
- input files can be treated as a sequence of lines

Example for writing:

```
from TextFile import TextFile

file = TextFile('an_example.gz', 'w')
file.write('This is the first line.\n')
file.write('This is the second line.\n')
file.write('This is  ')
file.write('the last line')
file.write(' of this file.\n')
file.close()
```

And for reading:

```
from TextFile import TextFile

for line in TextFile('an_example.gz'):
    print line[:-1]
```

Output formatting

Main tools: string operations
Built-in: string concatenation, string indexing, number conversion

Example: transform a list of number pairs into a string

```
s = ''
for pair in list:
    s = s + `pair[0]` + ' ' + `pair[1]` + '\n'
```

Or with an explicit format for real numbers:

```
s = ''
for pair in list:
    s = s + '%7.2f  %7.2f\n' % tuple(pair)
```

Output formatting with the percent operator is almost identical to the C function `printf` and similar to output with format specifications in Fortran. See the Python manual for a detailed description of formatting options.
More useful operations (justification, upper-/lowercase conversion) are implemented in the standard library module `string`; see the Python library manual for details.

Parsing input files

Main tasks:

- Finding the relevant information
- Converting it to convenient Python data structures

Most common first step: split the line into "words":

```
import string
words = string.split(line)
```

This splits the line at each white space region and returns a list of the word strings with surrounding spaces removed.
For finding specific key words, use string comparison and, if desired, case folding:

```
if string.lower(words[0]) == 'result:':
    result = string.atof(words[1])
```

The functions `atof` and `atoi` in the module `string` convert strings to real numbers and integers.
More complicated search methods are rarely needed, but are available in the forms of *regular expressions*; see the Python library manual for details.

Example: Read the output protocol of some iterative procedure and collect the data from lines of the form "result of step N: X" in a list.

```
from TextFile import TextFile
import string

data = []
for line in TextFile('output'):
    words = string.split(string.lower(line))
    if words[:3] == ['result', 'of', 'step']:
        data.append(string.atof(words[4]))
```

Example: Return everything between the lines containing "Matrix:" and "END" as an array.

```
from TextFile import TextFile
import Numeric, string

data = []
in_matrix = 0
for line in TextFile('output'):
    if line[:3] == 'END': break
    if line[:7] == 'Matrix:':
        in_matrix = 1
    elif in_matrix:
words = string.split(line)
numbers = map(string.atof, words)
data.append(numbers)
data = Numeric.array(data)
```

Fortran-style fixed-format files

Fortran programs treat text files very differently: items are identified by their position in the line, not by surrounding space. The layout of a line is defined by a *format specification*, such as `'A3,2I5,1X,4F15.4'`.
The module `FortranFormat` defines two classes to deal with such files. They allow Fortran-style input and output using the commonly used format specifications (A, D, E, F, G, I, and X formats, plus string constants for output). Repetition of individual formats and groups in parentheses is supported.

Input: from string to data

```
from FortranFormat import FortranFormat, FortranLine
s = '   59999'
format = FortranFormat('2I4')
line = FortranLine(s, format)
```

The result is a special sequence object containing the data in appropriate types. In the example, `line` has two integer elements with values 5 and 9999.

Output: from a list of data to a string

```
from FortranFormat import FortranFormat, FortranLine
format = FortranFormat('2D15.5')
line = FortranLine([3.1415926, 2.71828], format)
print str(line)
```

Note: The second argument of `FortranLine` can also be the original format string. However, for repeated use of the same format this is much slower, because parsing the format string is expensive.

Binary files

Caution: *The format of binary files often depends on the machine being used and in the case of Fortran files also on the compiler or run-time library.*

Binary files require a different mode specification: `'rb'` for reading, `'wb'` for writing. I/O still works via strings, which are interpreted as a sequence of bytes.

Note: On some operating systems there is no real difference between `'r'` and `'rb'` or `'w'` and `'wb'`. But the correct form is needed to ensure portability.

Output: as for text files using `file.write(string)`.
Input:

```
file = open('a_binary_file', 'rb')
data1 = file.read(4)  # read four bytes
data2 = file.read(11) # read 11 bytes
rest = file.read()    # read the rest of the file
file.close()
```

Each read operation reads *at most* the number of bytes indicated, but less if the end of the file has been reached.
Conversion between byte sequences in strings and Python data objects is handled by the module `struct`. It uses a *format string* to interpret the binary data, in which each letter stands for one binary data object (`'i'` for integer, `'f'` for float, ...). See the Python library manual for a full list.

Caution: *To be precise, each letter stands for a C data type, e.g.* `'i'` *stands for the C type* `int`*. The number of bytes corresponding to the C type depends on the machine and compiler being used.*

Conversion from data objects to a binary string: `struct.pack(format, data1, data2, ...)`
Conversion from a binary string to a tuple of data objects: `struct.unpack(format, binary_string)`

Example: Read a Fortran binary file and print the length (in bytes) of each record, assuming the file layout used by most Unix compilers.

```
import struct

file = open('a_fortran_binary_file', 'rb')
while 1:
    # read the record length field
    data = file.read(4)
    if not data: break
    record_length = struct.unpack('l', data)[0]
    print "Record of length ", record_length
    # skip the rest of the record
    data = file.read(record_length+4)
file.close()
```

Standard file formats

Some text and binary file formats are widely used. They can be used very efficiently if a module is written that presents the data to its users in a convenient way.
Examples:

- The module `PDB` reads and writes macromolecular configurations in the format of the Protein Data Bank. It presents the data to client code in a hierarchy of objects: chains – residues – atoms. These objects can easily be inspected or changed without detailed knowledge of the file format.
- The module `VRML` writes 3D scenes in the Virtual Reality Modeling Language. Client modules specify the objects by geometrical and visual information, e.g. "a green sphere of radius 5 at point (1., 2., 0.)".

Further advantage: compatible modules can encode the same information in different standard formats.

netCDF files

The netCDF format is a popular format for binary files. It is portable between machines and self-describing, i.e. it contains the information necessary to interpret its contents. A free library provides convenient access to these files.

The module `netcdf` provides a Python interface to the netCDF library which presents the data in the form of objects that behave very much like arrays.
A netCDF file contains any number of *dimensions* and *variables*, both of which have unique names. Each dimension has a value (a positive integer), and each variable has a shape defined by a set of dimensions, and optionally attributes whose values can be numbers, number sequences, or strings. One dimension of a file can be defined as “unlimited”, meaning that the file can grow along that direction. Some attributes are standardized by convention, e.g. the attribute `unit` which defines physical units. There are also global attributes attached to the file instead of individual variables.
The Python interface defines a file and a variable object type. Their netCDF attributes become Python attributes. File objects also have two special attributes, `dimensions` and `variables`, whose values are dictionaries. Variable objects are accessed by indexing and support all index options of arrays. The result of an indexing operation is an array.
The unlimited dimension needs a special treatment; standard arrays don’t change their size. The shape of a variable object always reflects the current size; it is updated whenever the shape attribute is explicitly requested. Indexing operations use the information from the last shape enquiry.

Example: writing a netCDF file

```
from netcdf import NetCDFFile
import Numeric

# Create a file
file = NetCDFFile('test.nc', 'w')
# Set the title
file.title = "Demo file"
# Create two dimensions
file.createDimension('n1', 10)
file.createDimension('n2', 20)
# Create a variable and set it to zero
foo = file.createVariable('foo', Numeric.Float,
                          ('n1', 'n2'))
foo[:] = 0.
# Close the file
file.close()
```

Example: print all variables in a netCDF file

```
from netcdf import NetCDFFile

file = NetCDFFile('test.nc', 'r')
for name, var in file.variables.items():
    print "Name: ", name
    print "Shape: ", var.shape
    print "Value: "
    print var[:]
file.close()
```

Plotting and visualization

Plotting via Gnuplot

The module `Gnuplot` provides a very basic interface to the public-domain plotting program Gnuplot. It is meant for interactive plotting jobs, not for production-quality plotting.
The function `plot` produces a 2D line plot in a newly opened window. It takes any number of arguments, each of which is one dataset. All datasets are plotted together.
A dataset can be any sequence of points, and a point is either a number (y value) or a pair (x, y) of numbers. If there are no x values in the dataset, they are assumed to be the integers from 1 to the number of points. Typical dataset descriptions are lists of numbers, lists of lists/tuples, or arrays.
An optional keyword argument `file=`*filename* generates the plot in the named file in PostScript format, rather than opening a window.

Example:

```
from Gnuplot import plot
import Numeric;  N = Numeric

x = 0.1*N.arrayrange(100.)
y = N.sin(x)
dataset = N.transpose(N.array([x, y]))
plot(dataset)
```

Visualization with VRML

The Virtual Reality Modeling Language (VRML) is a standard format for describing arrangements of 3D objects with various visual properties. VRML viewers are available for all common platforms, and many of them are free. This makes VRML a good solution for flexible low-cost visualization.
The module `VRML` contains definitions of various geometric objects that can be put together to form a VRML scene. There are also definitions for common colors and for color scales. Assembled scenes can be written to a VRML file (which can be compressed) or fed directly to a VRML viewer, whose name must be indicated by the environment variable `VRMLVIEWER`.
The available VRML objects are:

- `Sphere(center, radius)` creates a sphere.
- `Cube(center, edge)` creates a cube centered around a given point. The edges are oriented along the x, y, and z axes.
- `Cylinder(point1, point2, radius)` creates a cylinder whose axis runs from `point1` to `point2` with the radius specified.
- `Cone(point1, point2, radius)` creates a cone whose base is a circle around `point1` with radius `radius` and whose tip is at `point2`.
- `Line(point1, point2)` creates a line from `point1` to `point2`.
- `Arrow(point1, point2, radius)` creates an arrow from `point` to `point2` with a shaft radius of `radius`.

All objects can have one or more properties that affect their appearance. The most important property is the *material*, which defines color, transparency, reflectivity, etc. In this tutorial, only *diffuse materials* in various predefined colors are used, i.e. non-transparent non-reflecting materials. See a book on VRML for the full possibilities.
The material is specified with the keyword argument `material=`*material*. The value is a object of type `Material`, e.g. created by `DiffuseColor(color_name)`.

Example: display a black line through a sequence of points

```
import VRML
from Vector import Vector
```

```
import Numeric;  N = Numeric

# Make points that form a spiral
points = []
for t in N.arrayrange(0., 10., 0.1):
    x = N.sin(2*t)
    y = N.cos(2*t)
    z = t
    points.append(Vector(x, y, z))

# Create an empty scene and black material
scene = VRML.Scene([])
line_material = VRML.DiffuseMaterial('black')

# Create the lines and put them into the scene
for i in range(len(points)-1):
    scene.addObject(VRML.Line(points[i],
                              points[i+1],
                              material=line_material))

# View the scene
scene.view()
```

Example: display a scalar function on a grid by color coding

```
import VRML
from Vector import Vector
import Numeric;  N = Numeric

# Define grid and calculate function
grid = N.arrayrange(10.)
values = N.sin(grid) * \
         N.sin(grid[:, N.NewAxis]) * \
         N.cos(grid[:, N.NewAxis, N.NewAxis])

# Find range and define color scale
high = N.maximum.reduce(N.ravel(N.fabs(values)))
scale = VRML.SymmetricColorScale(high)
```

```
# Create scene and objects
scene = VRML.Scene([])
for i1 in range(len(grid)):
    for i2 in range(len(grid)):
        for i3 in range(len(grid)):
            x = Vector(grid[i1], grid[i2], grid[i3])
    m = VRML.Material(diffuse_color = scale(values[i1, i2, i3]))
            scene.addObject(VRML.Sphere(x, 0.2, material = m))

# View
scene.view()
```

Interfacing to external programs and subroutine libraries

Often existing code must be incorporated into a Python program.
Strategies:
For executable programs: generate input files in Python, run external program, parse output files. Wrap everything in a class or function.
For subroutine libraries: write (or generate with suitable tools) a Python interface module (C module) that is linked with the library.

Wrapping external programs

Temporary files
Files used for passing information between programs should be temporary and deleted automatically at the end. Problem: we need a filename which is guaranteed to be unused.
The module `tempfile` contains the function `mktemp` which returns a filename in a special directory for temporary files which does not correspond to any existing file.
At the end of the processing, the file can be deleted by calling the function `unlink` in the module `os`.

Running programs
The module `os` provides several functions that start external programs.

Caution: *Running external programs involves operating system calls that may not be available on all systems. The examples have been tested on Unix systems only.*

The function `os.system(command)` executes the command (a string) as if it had been typed into a command interpreter (`/bin/sh` on Unix systems). The command can contain input and output redirections and even run several programs in a pipe. The function will return when the external command finishes.

The function `os.popen(command, mode)` also executes the specified command by passing it to the command interpreter. However, the command runs in parallel to the Python process and is linked to it by a pipe, which is the return value of the function. If the mode is `'w'`, the Python program can *write* to the command and thereby provide input to it. If the mode is `'r'`, the Python program can *read* the output of the program from the pipe.

Caution: *These functions should be used with care. The commands can be anything, including* `'/bin/rm -rf *'`*. There is no protection against mistakes.*

Example: pass a text string to an external editor and return the modified version

```
import os
import tempfile

editor = 'xedit'

def edit(text):
    filename = tempfile.mktemp()
    file = open(filename, 'w')
    file.write(text)
    file.close()
    os.system(editor + ' ' + filename)
    file = open(filename)
    text = file.read()
    file.close()
    os.unlink(filename)
    return text
```

Running programs in the background
Often external programs should continue to run in parallel to the Python process, for example visualization programs.
Starting a parallel process is a four-step procedure:

1. Create a second process with `os.fork()`. The second process will also run the Python program, but will receive a different return value.
2. Use `os.system` or `os.popen` in the second process to run the external program.
3. Clean up, e.g. delete temporary files.
4. Terminate the second process with `os._exit(0)`.

Example: run the program defined by the environment variable VRMLVIEWER in the background:

```
import os
vrml_file = 'temp.wrl'
if os.fork() == 0:
    os.system(os.environ['VRMLVIEWER'] + ' ' + vrml_file +
              ' 1> /dev/null 2>&1')
    os.unlink(vrml_file)
    os._exit(0)
```

Wrapping libraries by C extension modules

Python modules can be written in C instead of Python, such modules are called *C extension modules*. They are covered in detail in other tutorials. C extension modules can be used to provide access to existing subroutine libraries from Python. This approach has been used for LAPACK and FFTPACK in the numerics extension.
There are two approaches for generating C extension modules to wrap libraries:

- Write the extension module manually. This offers maximum flexibility and opportunities for optimization, but is doable only for small modules.

- Use an automatic interface generator to generate the C module.

The most capable interface generator for Python is called SWIG and is covered in detail by a separate tutorial.

The following example (from the SWIG manual) shows how a small C library is wrapped by an extension module. The C library is:

File example.c

```
#include <time.h>

double My_variable = 3.0;

int fact(int n) {
  if (n <= 1) return 1;
  else return n*fact(n-1);
}

int mod(int n, int m) {
  return (n % m);
}

char *get_time() {
    long            ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

The SWIG input file specifies the name of the module to be generated and the prototypes of the C functions and variables:

File example.i

```
%module example

extern double My_variable;
extern int    fact(int);
extern int    mod(int n, int m);
extern char   *get_time();
```

Typing

```
swig -python example.i
```

generates the interface module in the file example_wrap.c. This can now be compiled like any other C extension module.
Assuming that shared libraries are supported, the compilation procedure requires only a small module specification file:

File Setup

```
*shared*
example example_wrap.c example.c
```

Also needed is the file Misc/Makefile.pre.in from the Python distribution. Two more commands create a ready-to-load dynamic library:

```
make -f Makefile.pre.in boot
make
```

The new module can be used immediately, e.g. by running the following test program:

File example.py

```
from example import *
print get_time()
print "My Variable = ", cvar.My_variable
for i in range(0,14):
    n = fact(i)
    print i, "factorial is ", n

for i in range(1,250):
    for j in range(1,250):
        n = mod(i,j)
        cvar.My_variable = cvar.My_variable + n

print "My_variable = ", cvar.My_variable
```

Fortran libraries
Most operating systems/compilers allow to mix C and Fortran code in programs. This makes it possible to call Fortran libraries from C extension modules and hence wrap Fortran libraries into Python.

Caution: *The details of mixed C/Fortran programming depend on the compiler Check your Fortran compiler manual for relevant information.*

There is currently no automatic interface generator for Fortran libraries; the C tools must be used. Two alternatives:

- Construct C prototypes that are compatible with your Fortran libraries and feed them to SWIG. The prototypes will be compiler dependent.
- Write a C wrapper around your Fortran code and apply SWIG to it. The added C layer can be used to hide compiler dependencies (via the C preprocessor) and to modify arguments to the Fortran code if that seems appropriate.

Writing C modules for efficiency

Another use for C extension modules is the implementation of time-critical code that would be too slow in Python.

Note: Time-critical code typically makes up only a small part of a complete program (5-10%). There is no reason to write a large program completely in a low-level language just because a small part requires it. Mixed-language Python/C programming offers many advantages.

C extension modules written from scratch for integration in Python code can use the interpreter as a utility library for various operations (I/O, error handling, memory allocation, etc.).

Profiting from dynamic libraries
On most systems C extension modules are compiled into dynamic libraries which are loaded when the module is imported. This permits *dynamic C code generation*: a Python module can generate or modify a C program file, compile it into a dynamic library, and immediately import and execute it.
Applications:

- Special-purpose compilers for efficiency

- Automatic recompilation of libraries that depend on compile-time parameters (e.g. Fortran libraries with static dimensions).
- Run-time choice between different implementations.

Scientific libraries and applications

Plotting and graphics:

- An interface to the plotting library GIST (part of the Yorick package, see ftp://icf.llnl.gov/pub/Yorick/) is available at ftp://icf.llnl.gov/pub/python/gistmodule.tar.gz.
- The DISLIN plotting package at http://www.mpae.gwdg.de/dislin/dislin.html comes with a Python interface for some systems (binaries for Python 1.4 only, no source code).
- The Python Imaging Library (PIL) provides pixel-oriented graphics operations. See http://www.python.org/sigs/image-sig/Imaging.html.
- An OpenGL interface permits the generation of 3D visualizations. See http://www.python.de/.

Mathematical and scientific libraries and applications:

- An interface to the Khoros package is available at http://windchime.arc.nasa.gov/˜grendel/pkim.html.
- An interface to the Simple Algebraic Math Library (SAML) is available at ftp://topo.math.u-psud.fr/pub/bousch/.
- The Molecular Modeling Toolkit (MMTK)is available at http://starship.skyport.net/crew/hinsen/mmtk.html.

Check also the collection of mathematical code at http://www.python.org/ftp/python/contrib/Math/INDEX.