Python for Scientific Computing 2

Konrad Hinsen
Centre de Biophysique Moléculaire, CNRS
Rue Charles Sadron; 45071 Orléans
E-Mail: hinsen@cnrs-orleans.fr

9 October 2003

Overview

- netCDF files
- Object-oriented programming
- Error handling
- Running external programs
- Integrating Fortran libraries
- Integrating C libraries

1 netCDF files

Features:

- store multiple multidimensional arrays
- different data types
- binary format, yet portable

- self-describing
- efficient access to sub-arrays
- can grow along one dimension

2 netCDF files

Python view: "arrays on disk"

netCDF file objects: represent a file with variables, dimensions, attributes, ...

netCDF variable objects: represent the data, act as much as possible like arrays

3 Writing netCDF files

```
from Numeric import *
from Scientific.IO.NetCDF import *
ncfile = NetCDFFile('test.nc', 'w',
                    'netCDF demonstration')
ncfile.createDimension('natoms', 100)
ncfile.createDimension('xyz', 3)
ncfile.createDimension('t', None)
time = ncfile.createVariable('time', Float,
                             ('t',))
time.units = 'ps'

conf = ncfile.createVariable('conf', Float,
                          ('t','natoms','xyz'))
conf.units = 'nm'

for i in range(10):
    time[i] = i*0.001
    for j in range(100):
        conf[i, j] = array([0., 0., 0.])
```

```
ncfile.close()
```

4 netCDF file inspection

Output of `ncdump -h test.nc`:

```
netcdf test {
dimensions:
        natoms = 100 ;
        xyz = 3 ;
        t = UNLIMITED ; // (10 currently)
variables:
        double time(t) ;
                time:units = "ps" ;
        double conf(t, natoms, xyz) ;
                conf:units = "nm" ;

// global attributes:
                :history = "netCDF demonstration" ;
}
```

5 Reading netCDF files

```
from Numeric import *
from Scientific.IO.NetCDF import *
ncfile = NetCDFFile('test.nc')
time = ncfile.variables['time']
conf = ncfile.variables['conf']
for j in range(100):
    print "Atom", j
    trajectory = conf[:, j, :]
    print trajectory
ncfile.close()
```

6 Object-oriented programming

Simple idea: Define data types that fit your problem.

Advantages:

- Data types are easier to isolate than parts of algorithms.
- Data types are more stable than algorithms as code evolves.
- Data types encapsulate implementation details.

7 Example: complex numbers

```
class Complex:

    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return Complex(self.real+other.real,
                       self.imag+other.imag)

    def conjugate(self):
        return Complex(self.real, -self.imag)

    def __str__(self):
        return 'Complex(%f, %f)' % (self.real, self.imag)
```

8 Example: complex numbers

```
a = Complex(1., 2.)
b = Complex(-2., 3.)
print a.conjugate() + b
```

9 Object-oriented design

Central question: What are the "right" object types in a non-trivial code?

- Requires practice.
- Trial and error.
- Don't hesitate to rewrite code for better design, it always pays off in the long run.

10 Object-oriented design

Candidates for object types:

1. **Mathematical objects**
   - numbers
   - matrices
   - vectors
2. **Physical objects**
   - particles (atoms, electrons, ...)
   - wave functions
   - pseudo-potentials
3. **Abstract objects**
   - algorithms (e.g. integrators)
   - object factories

11 Polymorphism

Idea: Different object types can implement similar operations using the same names. Algorithms can then be implemented once for any of these types.

12 Example: atoms

```
class Atom:

    def __init__(self, element, position):
        self.element = element
        self.position = position

    def translateBy(self, vector):
        self.position += vector

    def centerOfMass(self):
        return self.position

    def mass(self):
        return masses[self.element.lower()]
```

13 Example: molecules

```
class Molecule:

    def __init__(self, atom_list):
        self.atoms = atom_list

    def translateBy(self, vector):
        for atom in self.atoms:
            atom.translateBy(vector)

    def centerOfMass(self):
        sum1 = 0.
        sum2 = Vector(0., 0., 0.)
        for atom in self.atoms:
            sum1 += atom.mass()
            sum2 += atom.mass()*atom.position
        return sum2/sum1
```

14 Inheritance

Idea: Often one object type is a specialization of another one. It can **inherit** the general operations and add its own ones.

15 Example: peptide chains

```
class PeptideChain(Molecule):

    def __init__(self, residue_list):
        self.residues = residue_list
        atoms = []
        for r in residue_list:
            atoms = atoms + r.atoms
        Molecule.__init__(self, atoms)

    def __getitem__(self, index):
        return self.residues[index]
```

16 Error handling

Problem: some function can't do what it is supposed to do. How to signal this to its clients?

Usual solution: return an error code

Difficulties:

- need a "special" return value
- error has to be passed up the calling chain

In practice: a lot of messy error-handling code.

17 Exceptions

Python approach: A special error-signalling channel, independent from return values.

When something goes wrong, signal an "exception":

```
def logarithm(x):
    if x <= 0:
        raise ValueError, 'not positive'
    return log(x)
```

18 Exceptions

Application:

```
>>> print logarithm(2.)
0.69314718056
>>> print logarithm(0.)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in logarithm
ValueError: not positive
```

Python shows the **calling chain** and stops. At this point, the problem can be analyzed with an interactive debugger.

19 Exceptions

Programs can **catch** and **handle** exceptions:

```
try:
    y = logarithm(x)
except ValueError:
    print 'x was not positive'
```

Exceptions can be handled at any point in the calling chain.

20 Running external programs

Executing a shell command line:

```
import os
os.system('program arg1 arg2 < file1 > file2')
```

Feeding input to a program:

```
import os
stdin = os.popen('program arg1 arg2', 'w')
stdin.write('input line 1\n')
stdin.write('input line 2\n')
stdin.close()
```

21 Running external programs

Getting output from a program:

```
import os
stdout = os.popen('program arg1 arg2', 'r')
while 1:
    line = stdout.readline()
    if not line: break
    print line
stdout.close()
```

22 Running external programs

Handling input and output of a program:

```
import os
stdin, stdout = os.popen2('program arg1 arg2')
stdin.write('input data')
```

```
stdin.close()
while 1:
    line = stdout.readline()
    if not line: break
    print line
stdout.close()
```

Note: synchronisation is not easy!

23 Background execution

```
import os
if os.fork() == 0:
    os.system('program')
    os._exit(0)
# go on with program execution here
```

24 Calling Fortran routines

Example routine (Fortran 77):

add.f

```
      real function add(array, n)
      integer n
      real array(n)
      integer i
      real sum
      sum = 0.
      do 100 i = 1, n
         sum = sum + array(i)
100   continue
      add = sum
      end
```

25 Calling Fortran routines

Pyfort interface definition:

add.pyf

```
real function add (array, n)
integer n = size(array)
real array (n)
end
```

Run `pyfort add` and then

```
from add import add
print add([1, 2, 3])
```

26 Pyfort caveats

- no support for Fortran 90 features
- works only for "known" compilers (many)
- indexing issues for multidimensional arrays
- data type correspondance is compiler dependent

⇒ Read the manual!

27 Calling C routines

Example routine:

add.c

```
double add(double *array, int n) {
  int i;
  double sum = 0.;
  for (i = 0; i < n; i++) {
    sum += array[i];
  }
  return sum;
}
```

28 Calling C routines

SWIG interface definition:

add.i

```
%module add

%typemap(python,in) (double *a, int n) {
  $2 = (int)((PyArrayObject *)($input))->dimensions[0];
  $1 = (double *)((PyArrayObject *)($input))->data;
}

%inline %{
#include "Numeric/arrayobject.h"
%}

extern double add(double *a, int n);
```

29 Calling C routines

add_setup.py

```
from distutils.core import setup, Extension

add_module = Extension('add',
                       ['add_wrap.c', 'add.c'])
```

```
setup (ext_modules = [add_module])
```

Run

```
swig -python add.i
```

and then

```
python add_setup.py
```