

When all else fails: do-it-yourself programming

Konrad Hinsen
Laboratoire Lon Brillouin (CEA-CNRS)
CEA Saclay
91191 Gif sur Yvette
France

20 July 2004

Biomolecular simulations

Standard tasks:

- Run an MD simulation.
- Calculate atomic fluctuations.
- Visualize trajectory.
- ...

⇒ standard programs

Nonstandard tasks:

- Automatize a multistep operation.
- Extract data from a file.
- File format conversion.
- Implement new algorithms.
- Nonstandard visualization.
- ...

⇒ **do-it-yourself programming**

Programming options

Shell scripts:

- quick to write
- work on any (Unix) system
- very limited possibilities
- difficult to read
- very slow execution

Suitable for: automatization of simple operations

Low-level languages (C, C++, Fortran, ...):

- fast execution
- good development tools
- require significant experience
- long development times
- can be difficult to read

Suitable for: time-critical algorithms

Programming options

High-level languages (Perl, Python, Ruby, ...):

- complete programming languages
- easy to learn
- fast development
- easy to read
- interface to low-level languages
- slow execution

Suitable for: everything that is not time-critical

Empirical rule: 90% of a program is *not* time critical

Therefore:

- Everyone should use a high-level language.
- Developers of numerical methods must also know a low-level language.

High-level languages

Features:

- Interpreted (no compilation)
- High-level data types (lists, dictionaries, ...)
- No variable type declarations
- Automatic memory management
- Large library of existing code

Applications:

- Complete programs (file format conversion, ...)
- “Glue” between other programs
- “Scripting” of library code
- Mixed high/low-level programs

Python

Features:

- Clean syntax
- Object-oriented
(→ problem-oriented data structures)
- Good low-level language interface
(C, C++, Fortran)
- Free and portable implementation
- Well established as a scripting language for computational science

Examples:

- Sum up numbers in the third column of a text file

```
sum = 0.  
for line in file('my_data'):  
    sum = sum + float(line.split()[2])  
print sum
```

Python

Examples:

- Calculate the radius of gyration of a protein

```
from MMTK.Proteins import Protein
import Numeric

protein = Protein('protein.pdb')
center = protein.centerOfMass()
r_sq = 0.
for atom in protein.atomList():
    mass = atom.mass()
    distance = atom.position()-center
    r_sq = r_sq + mass*distance*distance
r_sq = r_sq/protein.mass()
print Numeric.sqrt(r_sq)
```

Modules

Python code is structured into *modules*. Objects in another module must be *imported* before they can be used. One can import the module, individual objects from a module, or everything from a module.

Examples:

```
import Numeric
print Numeric.sin(Numeric.pi/3)
```

```
from Numeric import pi, sin
print sin(pi/3)
```

```
from Numeric import *
print sin(pi/3)
```

Modules can contain other modules:

```
from Scientific.Geometry import Vector
print Vector(1., -2., 0.5).length()
```

```
from Scientific import Geometry
print Geometry.Vector(1., -2., 0.5).length()
```

```
import Scientific.Geometry
print Scientific.Geometry.Vector(1., -2., 0.5).length()
```

Basic math

Numbers

Integer 0, 1, 2, 3, -1, -2, -3
Real 0., 3.1415926, -2.05e30, 1e-4
Imaginary/Complex 1j, -2.5j, 3+4j
Special numbers: Numeric.pi, Numeric.e

Arithmetic

Addition 3+4, 42.+3, 1+0j
Subtraction 2-5, 3.-1, 3j-7.5
Multiplication 4*3, 2*3.14, 1j*3j
Division 1/3, 1./3., 5/3j
Power 1.5**3, 2j**2, 2**-0.5

Functions

Numeric.sqrt, Numeric.log, Numeric.log10, Numeric.exp, Numeric.sin,
Numeric.cos, Numeric.tan, Numeric.arcsin, Numeric.arccos, Numeric.arctan,
Numeric.sinh, Numeric.cosh

Text strings

Two forms: 'abc' or "abc"

Multiline strings:

```
'''This string extends  
over two lines.'''
```

Control characters: 'end of line:\n'

Concatenation: 'abc' + 'def' gives 'abcdef'

Repetition: 3*"ab" gives "ababab"

Indexing: 'abc'[0] is 'a', 'abc'[1:] is 'bc'

Conversion to numbers: int('12'), float('-2.1')

Decomposition into words: 'one two three'.split()

... and many other string operations, check the manual!

Lists

Lists are sequences of arbitrary objects:

```
some_prime_numbers = [2, 3, 5, 7, 11]  
names = ['Smith', 'Jones']  
a_mixed_list = [3, [2, 'b']]
```

Elements and subsequences are accessed by indexing:

```
print names[0]
names[1] = 'Python'
print a_mixed_list[-1]
some_prime_numbers[3:] = [17, 19, 23, 29]
```

Concatenation: `[1, 2]+'a'` gives `[1, 2, 'a']`

Repetition: `3*[0]` is `[0, 0, 0]`

Some more list operations:

```
names.append('van Rossum')
some_prime_numbers.sort()
a_mixed_list.reverse()
print len(names)
print names.index('Smith')
```

Dictionaries

Dictionaries map (almost) arbitrary *keys* to arbitrary *values*.

Examples:

```
atomic_mass = {'H': 1., 'C': 12., 'S': 32.}
atomic_mass['O'] = 16.
print atomic_mass['O'] + 2*atomic_mass['H']
```

Lookup with default value:

```
atomic_mass.get('X', 0.)
```

Applications:

- Storing tables
- Counting (counted objects are keys!)
- Classifying (values are lists)
- ...

Flow control

Conditions:

```
if a > b:
    print 'a > b'
elif a == b:
    print 'a == b'
else:
    print 'a < b'
```

Loops over sequences (lists, strings, ...):

```
for x in [1.2, 2.5, -0.2]:
    ...
for n in range(10):
    ...
```

Conditional loops:

```
while n > 0:
    ...
    n = n - 1
```

The block structure is defined by *indentation*.

Functions

```
def square(x):
    return x**2
```

```
print square(3)
```

Multiple return values:

```
def sum_and_difference(a, b):
    return a+b, a-b
```

```
x, y = sum_and_difference(1, 2)
```

Variables created in a function are *local*:

```
def square(x):
    sq = x**2
    return sq
sq = 5.
print square(1.)
print sq
```

prints 5., not 1.

Error handling

```
def inverse(x):  
    return 1./x  
print inverse(0)
```

Output:

```
Traceback (innermost last):  
  File "demo.py", line 3, in ?  
    print inverse(0)  
  File "demo.py", line 2, in inverse  
    return 1./x  
ZeroDivisionError: float division
```

Intercepting an error:

```
try:  
    print inverse(0)  
except ZeroDivisionError:  
    print 'Division by Zero'
```

Object-oriented programming

Programming by creating *problem-specific* data types and *operations* for these data types.

Example: a simple representation for atoms

```
from Scientific.Geometry import Vector  
  
class Atom:  
  
    def __init__(self, name, position):  
        self.name = name  
        self.position = position  
  
    def translateBy(self, vector):  
        self.position = self.position+vector  
  
atom = Atom('C-alpha', Vector(1., 0., 0.))  
atom.translateBy(Vector(0.1, -0.2, 0.05))
```

Procedural approach:

```
from Scientific.Geometry import Vector

atom = ['C-alpha', Vector(1., 0., 0.)]
atom[1] = atom[1] + Vector(0.1, -0.2, 0.05)
```

Problem: code that works on atom data must know its representation.

Object-oriented programming

Advantages of OOP:

- Encapsulation: users of a class don't need to know its implementation. All code that knows how atom data are stored is located in the class `Atom`.
Advantage: the implementation can be changed without affecting client code.
- Data representations are more universal than procedures.
Advantage: classes can be reused for other projects.
- Classes directly represent real-world objects.
Advantage: code becomes easier to understand.

Object-oriented programming

Functions defined in classes are called *methods*. The class of an object determines which method is called (*polymorphism*).

Suppose we also have a `Molecule` class:

```
class Molecule:

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

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

m = Molecule([Atom('O1', Vector(0., 0., 0.)),
              Atom('O2', Vector(0., 0.12, 0.))])
```

Then `x.translateBy(v)`

- calls `Atom.translateBy` if `x` is an atom

- calls `Molecule.translateBy` if `x` is a molecule
- reports an error if `x` is something else

Advantage: higher-level code can be written such that it works on data of different types.

Object-oriented programming

Inheritance is the definition of a data type as a *specialization* of an existing one.

```
class Ion(Atom):

    def __init__(self, name, position, charge):
        Atom.__init__(name, position)
        self.charge = charge
```

```
ion = Ion('Fe', Vector(0., 1., 0.), 2.)
ion.translateBy(Vector(-0.1, -0.3, 0.05))
```

An `Ion` inherits all the features of an `Atom`, and adds its own ones (the charge).

Advantage: several specializations of general classes can be written without changing existing (tested!) code and without duplicating code.

Profiting from existing code

Python standard library:

- string and text processing
- object serialization: write any object to a file
- thread support
- database interfaces
- profiling and debugging
- most internet protocols
- complete Web server
- access to common data and file formats, including HTML and XML

Other free libraries and packages:

- Numerical Python, Scientific Python
- MMTK (molecular simulations)
- plotting
- signal processing
- interfaces to scientific data formats
- distributed objects (CORBA etc.)
- Zope (Web server management system)