EMBO Practical Course on Biomolecular Simulations: Code writing

Konrad Hinsen
Centre de Biophysique Moléculaire (CNRS)
Orléans, France

8 July 2000

1 Getting started

1.1 Python

This practical will use a high-level programming language called Python, as well as some extensions to it that are useful for scientific applications. For more information, see

- Python home page: `http://www.python.org/`
- Python Tutorial: `http://www.python.org/doc/current/tut/tut.html`
- Python Library Reference: `http://www.python.org/doc/current/lib/lib.html`
- Numerical Python: `http://numpy.sourceforge.net/`
- Scientific Python: `http://starship.python.net/crew/hinsen/scientific.html`
- A more complete introduction to Python for scientists can be found at `http://starship.python.net/crew/hinsen/`
- A catalogue of available Python modules can be found at `http://www.vex.net/parnassus/`

To run any of the examples in this practical, do one of the following:

- Type "`python`" (without the quotes) and then type in the example code line by line. To exit from Python, type Ctrl-D.

- Type the example code into a file with an editor of your choice and then type "`python -i example.py`" (without the quotes), assuming that your file is called `example.py`. To exit from Python, type Ctrl-D. If you do not use the "`-i`", Python will exit immediately after executing the program.

More convenient methods of using Python exist, but these two one have the advantage of working on all computers without additional programs.
When typing Python code, watch out for the number of spaces at the beginning of a line. Python uses this information for determining the structure of a program, i.e. to find out which lines belong to a loop or to a conditional block. The precise number of spaces is not important, but within one block all lines must have the same number of initial spaces, i.e. the first characters must line up properly.
It is of course impossible to learn a full programming language in 90 minutes. Many, if not most, features of Python are not mentioned at all. Some of the examples presented here could be simplified using more advanced Python features. Nevertheless, after working on these exercises you should be able to solve real-life problems that occur when doing biomolecular simulations, and you should be able to extend your knowledge of Python by self-study.

1.2 Modules

Python code comes in two flavors: scripts and modules. Scripts are programs that are executed; all examples in this practical are scripts. Modules are code repositories that are used by scripts (or by other modules). The Python Library contains a large amount of modules that provide ready-to-use code for a vast range of tasks, from mathematical functions to complete Web servers. And more useful modules are available as add-on packages, including a special library for biomolecular simulations (see the practical on normal modes).
Before you can use code from a module, you must *import* it so that Python knows that you want to use it. The various ways to import something from a module will be illustrated for the module `Numeric`:
Importing individual functions:

```
from Numeric import sqrt
from Numeric import log
from Numeric import sin, cos, tan
```

Importing everything from a module:

```
from Numeric import *
```

This seems very convenient (and it is!), but it is also risky, because you don't know what you are importing. You might accidentally overwrite some important function, for example. So use this carefully, and only at the beginning of a script.
You can also import just the module:

```
import Numeric
```

and then use ”extended” names for the functions:

```
print Numeric.sqrt(2)
```

2 Simple calculations

Numbers and arithmetic operation in Python are used just like in any other programming language. Note that, just like in C or Fortran, division of two integers yields an integer, i.e. `1/2` is zero!
The module `Numeric` (part of Numerical Python) contains the standard mathematical functions (sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sin, cosh), the constants `pi` and `e`, an array data type, plus functions that manipulate arrays. The mathematical functions work on integers, floats, and complex numbers, as well as on arrays of numbers. `Numeric` is the central module for scientific applications, you will need it for most examples in this course.
The module `Scientific.Geometry` (part of Scientific Python) contains, amongst other things, a class `Vector` which represents 3-dimensional vectors as they are used in geometry.

Examples:
A quick calculation:

```
from Numeric import *
print (sin(pi/3)+sqrt(2)-log10(42))**2
```

Or equivalently:

```
import Numeric
print (Numeric.sin(pi/3)+Numeric.sqrt(2)-Numeric.log10(42))**2
```

Or use an abbreviation:

```
import Numeric
N = Numeric
print (N.sin(pi/3)+N.sqrt(2)-N.log10(42))**2
```

A simple geometrical problem, the distance between two points:

```
from Scientific.Geometry import Vector
p1 = Vector(0., 1., -1.)
p2 = Vector(1.5, -3.7, 2.1)
print (p1-p2).length()
```

Calculate an angle in a triangle:

```
from Scientific.Geometry import Vector
p1 = Vector(0., 1., -1.)
p2 = Vector(1.5, -3.7, 2.1)
p3 = Vector(0.7, 0.1, -0.5)
print (p1-p2).angle(p3-p2)
```

And a normalized vector perpendicular to a triangle:

```
from Scientific.Geometry import Vector
p1 = Vector(0., 1., -1.)
p2 = Vector(1.5, -3.7, 2.1)
p3 = Vector(0.7, 0.1, -0.5)
print ((p1-p2).cross(p3-p2)).normal()
```

Exercise: Write a program that calculates the area A of a triangle using the formula

$$A = \frac{1}{2} \left|(\mathbf{p}_1 - \mathbf{p}_2) \times (\mathbf{p}_3 - \mathbf{p}_2)\right|$$

3 Data structures: lists and strings

Lists may contain any kind of data, also mixed:

```
some_prime_numbers = [2, 3, 5, 7, 11, 13]
names = ['Smith', 'Jones']
a_mixed_list = [3, [2, 'b'], Vector(1, 1, 0)]
```

Lists elements and sublists can be retrieved or changed by indexing:

```
l = [2, 3, 5, 7, 11]
# first element (index is zero):
print l[0]
# second element (index is one):
print l[1]
# last element (negative indices count from end):
print l[-1]
# up to, but excluding, element number 3:
print l[:3]
# the last two elements:
print l[-2:]
# change first element to one:
l[0] = 1
```

The length of a list can be obtained by the function `len()`.
Many more list operations are available: inserting and removing elements, sorting, reversing, etc. Particularly important are *loops* over list elements:

```
for prime_number in [2, 3, 5, 7]:
    square = prime_number**2
    print square
```

The function `range(n)` returns a list of all integers from 0 to n-1. This is used for loops with an integer index:

```
for i in range(10):
    print i, i**2
```

Most of the operations for lists can be applied as well to any `sequence object`. A sequence object is an object that has elements which can be retrieved by integer indices. Examples are strings, arrays, and vectors.
Here is an example with strings:

```
text = 'EMBO course'
print text[0]
print text[:4]
print text[-6:]
print text[2:4]
```

Strings are sequences of characters. Unlike lists, they cannot be modified once they have been constructed. Many string operations are provided by the module `string`. The most important ones are:

- `split(string)` returns a list of the individual words in the string (defined by white space delimiters)
- `strip(string)` returns string with white space at the beginning and end removed
- `atoi(string)` converts a string to an integer
- `atof(string)` converts a string to an float

Any Python object (numbers, strings, vectors, ...) can be converted into a string by writing it in inverse apostrophes:

```
from Scientific.Geometry import Vector

a = 42
b = 1./3.
c = Vector(0, 2, 1)
print `a` + ' ' + `b` + ' ' + `c`
```

4 Manipulating text files

Reading from a text file:

```
file = open('filename')
lines = file.readlines()
file.close()
```

The variable `lines` now contains a list of strings, each containing one line from the file. The code can be reduced to one line:

```
lines = open('filename').readlines()
```

Here the file is closed automatically when the lines have been read.
For more convenient file handling, replace `open` by `TextFile`, to be imported from `Scientific.IO.TextFile`.

Note: The last character of each line is a newline character, written as `'\n'` in Python.

It is possible to read a file line by line, but reading everything at once is simpler and sufficient, unless very large files are treated.

Writing to a text file:

```
file = open('filename', 'w')
for name in ['Smith', 'Jones']:
    file.write(name + '\n')
file.close()
```

Example: The following program reads a file and prints the sum of all numbers in the second column.

```
import string

lines = open('filename').readlines()
sum = 0.
for line in lines:
    sum = sum + string.atof(string.split(line)[1])

print "The sum is:", sum
```

Exercise: Write a program that counts the number of lines and words in a file.

Example: The following program reads a PDB file and prints the number of carbon atoms. We identify a carbon atom as an atom whose name begins with C; this is *not* conforming to the PDB norm, but to the way many programs use PDB files.

```
import string

lines = open('protein.pdb').readlines()
carbons = 0
for line in lines:
    record_type = line[:6]
    if record_type == 'ATOM  ' or record_type == 'HETATM':
        atom_name = string.strip(line[12:16])
        if atom_name[0] == 'C':
            carbons = carbons + 1

print carbons, "carbon atoms"
```

Exercise: Modify this program to respect the PDB norm, according to which the first two characters of the atom name contain the element type and a space is written before one-letter element names.

5 Data structures: dictionaries

Dictionaries store associations between a set of keys and a set of values. They can be regarded as generalizations of sequences, because sequences are like dictionaries whose keys are a range of integers. Dictionaries allow almost any type of key, but define no order of their elements.
Dictionaries are accessed much like sequences: `dictionary[key]` returns the value associated with the key, and `dictionary[key]=value` changes it. There is also a handy way to look up an entry and return a default value if the key is not defined: `dictionary.get(key, default)`. A list of all keys is available with `dictionary.keys()`, and a list of all values with `dictionary.values()`. A dictionary entry can be deleted with `del dictionary[key]`.
The following code creates a dictionary containing the atomic masses of some chemical elements. It then adds another entry and finally calculates the mass of a water molecule.

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

A typical use for dictionaries is sorting data into categories. For example, the following program will read a PDB file and count the number of times each amino acid type occurs:

```
import string
```

```
residues = {}
for line in open('protein.pdb').readlines():
    record_type = line[:6]
    if record_type == 'ATOM  ' or record_type == 'HETATM':
        residue_name = line[17:20]
        residue_number = string.atoi(line[22:26])
        residue_dict = residues.get(residue_name, {})
        residues[residue_name] = residue_dict
        residue_dict[residue_number] = 1

for residue_name in residues.keys():
    print residue_name, len(residues[residue_name])
```

This program uses dictionaries at two levels: the dictionary `residues` has an entry for each residue type, and those entries are again dictionaries, which at the end of the loop contain a "one" (the value doesn't really matter) for each residue number of the respective type. This "one" is assigned again for each atom in the residue. This approach avoids the tedious procedure of determining residue boundaries.

Exercise: The XYZ format is a very simple (and limited) format for storing molecular configurations, which is used by several programs. The first line contains the number of atoms, the second line a comment, and the remaining lines contain one atom each, with four entries: first the element symbol (e.g. 'C' for carbon), and then the coordinates x, y, and z. The entries are separated by one or more spaces. This is an example for a single water molecule:

```
3
One water molecule
O 0.  0.     0.
H 0.  0.957  0.
H 0. -0.24  -0.927
```

Write a program that reads an XYZ file and calculates the center of mass of the atoms.

6 Visualization

Visualization is an essential tool in biomolecular simulations. Very elaborate visualization programs exist, which do a good job of presenting some commonly used kinds of data. However, it would frequently be useful to be able to construct specialised 3D visualizations for particular purposes. In the simplest case, this means adding arrows, axes, etc. to an image of a molecule. But one might also

want to construct completely different 3D scenes not related to molecular images at all.
Scientific Python provides three modules for scientific visualization. They differ only in generating input for different visualization programs: VRML 1.0 browsers VRML 97 browsers (also called VRML 2), and VMD. We will use VMD here because it is available for the practical. The module from which all graphics items are to be imported is then `Scientific.Visualization.VMD`.
The available 3D 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.

The following example program reads a PDB file and constructs a scene consisting of a blue sphere for each C-alpha atom and red lines connecting neighbouring C-alpha atoms:

```
from Scientific.Geometry import Vector
from Scientific.Visualization import VMD
import string

# Read the PDB file and make a list of all C-alpha positions
c_alpha_positions = []
for line in open('protein.pdb').readlines():
    record_type = line[:6]
    if record_type == 'ATOM  ' or record_type == 'HETATM':
        atom_name = string.strip(line[12:16])
        x = string.atof(line[30:38])
        y = string.atof(line[38:46])
        z = string.atof(line[46:54])
        if atom_name == 'CA':
            c_alpha_positions.append(Vector(x, y, z))
```

```
# Create the VMD scene and the materials and radii for spheres and lines
scene = VMD.Scene([])
c_alpha_material = VMD.DiffuseMaterial('blue')
c_alpha_radius = 0.3
line_material = VMD.DiffuseMaterial('red')

# Create the spheres and put them into the scene
for point in c_alpha_positions:
    scene.addObject(VMD.Sphere(point, c_alpha_radius,
                               material=c_alpha_material))

# Create the lines and put them into the scene
for i in range(len(c_alpha_positions)-1):
    point1 = c_alpha_positions[i]
    point2 = c_alpha_positions[i+1]
    scene.addObject(VMD.Line(point1, point2, material=line_material))

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

Exercise: Write a program that reads two PDB files containing two conformations of the same protein and produces a VRML scene containing an arrow from conformation 1 to conformation 2 for each C-alpha atom.