Python for Scientific Computing

Konrad Hinsen
Centre de Biophysique Moléculaire (CNRS), Orléans, France
and
Synchrotron Soleil, Saint Aubin, France

1 Python

The following links are good starting points for finding information about Python and Python code for scientific computing:

- Python home page: `http://www.python.org/`
- Python Tutorial: `https://docs.python.org/2.7/tutorial/index.html`
- Python Library Reference: `https://docs.python.org/2.7/library/index.html`
- Numerical Python/NumPy: `http://numpy.scipy.org/`
- Scientific Python: `http://dirac.cnrs-orleans.fr/ScientificPython/`
- SciPy: `http://www.scipy.org/`
- VPython: `http://www.vpython.org/`
- IPython: `http://ipython.scipy.org/moin/`
- matplotlib: `http://matplotlib.sourceforge.net/`
- Sphinx: `http://sphinx-doc.org/index.html`

For those with a Fortran background, this comparison between the two languages may be useful:

- `http://www.fortran90.org/src/rosetta.html`

2 Running Python programs

To run any of the examples in this course, 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 and you will not have the chance to inspect variables etc.

More convenient methods of using Python exist, but these two ones have the advantage of working on all computers without additional programs. The development environment IDLE is installed on many systems (try typing "idle" of you are on a Unix system) and very suitable for starting with Python programming. 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 non-blank characters must line up properly.
Most Python modules contain built-in documentation that can be accessed using the tool `pydoc`. For example, typing `pydoc numpy.array` will produce a short description of the function `array` in module `numpy`.
Finally, don't hesitate to experiment. It is often quicker to type a few test ocommands than to look up something in the manuals!

3 Python language and library overview

The following sections provide a brief summary of the Python language and of libraries that are important for scientific computing. This overview is of course far from exhaustive, but it is a useful references while working through the exercises.

3.1 Modules

Python code is structured by **modules**. Modules can contain functions, variables, classes, and other modules.
To use the contents of a module, they must be **imported**, either individually:

```
from module import function1, function2
```

or collectively:

```
from module import *
```

Modules can be grouped into **packages**:

```
from package.module1 import *
from package import module2
from package.module2 import function1
```

3.2 Numbers and arithmetic

Integer	0, 1, 2, 3, -1, -2, -3
Real	0., 3.1415926, -2.05e30, 1e-4
	(must contain dot or exponent)
Imaginary/Complex	1j, -2.5j, 3+4j
	(the last one is a sum)
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

3.3 Mathematical functions

The standard mathematical functions (`sqrt`, `log`, `log10`, `exp`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sin`, `cosh`), as well as the constants `pi` and `e`, are not part of the basic language, but contained in a module called `numpy`:

```
from numpy import sin, pi
print sin(pi/3.)
```

3.4 Text strings

3.4.1 Syntax

Two forms: `'abc'` or `"abc"`

Line breaks are indicated by `\n`: `"abc\ndef"`

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

Repetition: `3*"ab"` gives /"ababab"/

Triple quotes permit multi-line strings:

```
comment = '''This string
is split over
three lines.
'''
```

3.4.2 Operations

```
'ab cd e'.split()      ['ab','cd','e']
'1, 2, 3'.split(',')   ['1', ' 2', ' 3']
','.join(['1','2'])    '1,2'
' a b c  '.strip()     'a b c'
'text'.find('ex')      1
'Abc'.upper()          'ABC'
'Abc'.lower()          'abc'
```

Convert to number: `int('2')`, `float('2.1')`

Convert any data type to a string: `str(3)`, `str([1, 2, 3])`

3.5 Lists

Lists are very important in Python. They can store any kind of element:

```
some_prime_numbers = [2, 3, 5, 7, 11, 13]
names = ['Smith', 'Jones']
a_mixed_list = [3, [2, 'x'], '']
```

Concatenation	`[0,1]+['a','b']`
	$\rightarrow$ `[0,1,'a','b']`
Append	`names.append('Python')`
Sort	`names.sort()`
Reverse	`names.reverse()`

3.6 Tuples

Tuples are **immutable** lists. Once created, they can not be changed. They are more efficient than lists, and can be used in places where immutable objects are required.

Syntax: `(1, 2, 3)`

Note: single-element tuples require a comma: `(1,)`

3.7 Dictionaries

A dictionary stores a **mapping** from (almost) arbitrary **keys** to arbitrary **values**.

```
dictionary = {'a': 1, 'b': 2, 'c': 3}
dictionary['z'] = 26
print dictionary['c']
print dictionary.get('h', 'not found')
print dictionary.keys()
print dictionary.values()
print dictionary.items()
```

3.8 Indexing

Sequence objects (lists, strings, ...) permit **indexing**:

```
text = 'abc'
print text[1]
print text[-1]
```

and **slicing**:

```
print text[0:2]
print text[1:-1]
```

Note: The first element always has index `0`.
Mutable sequence objects (e.g. lists) permit **indexed assignment**:

```
names = ['Smith', 'Jones']
names[1] = 'Python'

some_prime_numbers = [2, 3, 5, 7, 11, 13]
some_prime_numbers[3:] = [17, 19, 23, 29]
```

3.9 Objects

An **object** is a collection of data. The **type** of an object determines what **operations** are available for it.

Type	Operations
Integer	`+ - * /`
List	`+ append sort` ...

Object-oriented programming: Programming by creating problem-specific object types

3.10 Variables

Fortran, C: A variable **stands for** a memory area holding data.

Python: A variable **refers to** a memory area holding data.
Two variables can refer to **the same** memory area.

```
a = [1, 2, 3]
b = a
a.append(4)
print b
```

prints `[1, 2, 3, 4]`

3.11 Control flow

3.11.1 Conditions

equal	`a == b`
not equal	`a != b`
greater than	`a > b`
less than	`a < b`
greater than or equal	`a >= b`
less than or equal	`a <= b`

The result of a comparison is `True` or `False`.

Logical operators: `not`, `and`, `or`

```
if a > 0:
   print 'greater'
elif a == 0:
   print 'equal'
else:
   print 'smaller'
```

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

3.11.2 Loops

```
for item in sequence:
    print item
```

Counting loop ("do loop"):

```
for i in range(5):
    print i
```

Note: `range(5)` yields the list `[0, 1, 2, 3, 4]`

3.12 File handling

Fortran: A file is a sequence of records, which can be "formatted" (text) or "unformatted" (binary).

C/Python: A file is a sequence of bytes. Text files use **control characters** for formatting (line breaks etc.)

3.12.1 Reading files

Read whole file into a string object:
```
contents = file('filename').read()
```

Read whole file into a list of lines:
```
contents = file('filename').readlines()
```

Read line by line:
```
for line in file('filename'):
    print line
```

3.12.2 Writing files

```
output_file = file('filename', 'w')
output_file.write('line1\n')
output_file.write('line2\n')
output_file.close()
```

Replace `'w'` by `'a'` to append to an existing file.

3.12.3 Binary files

The module `struct` provides conversion from string data to other data types and back.
Reading:

```
from struct import unpack
binary_data = file('data.bin').read(8)
a, b = unpack('ii', binary_data)
```

Writing:

```
from struct import pack
binary_data = pack('id', 1, 2.5)
file('data.bin', 'w').write(binary_data)
```

3.13 Functions

Like C, but unlike Fortran, Python makes no difference between functions and subroutines. Functions can have zero, one, or more return values.

```
import numpy as np

def distance(r1, r2):
    d = r1 - r2
    return np.sqrt(np.dot(d, d))

print distance(np.array([1., 0., 0.]),
               np.array([3., -2., 1.]))
```

Multiple return values:

```
import numpy as np

def cartesianToPolar(x, y):
    r = np.sqrt(x**2+y**2)
    phi = np.arccos(x/r)
    if y < 0.:
        phi = 2.*np.pi-phi
    return r, phi

radius, angle = cartesianToPolar(1., 1.)
```

Functions can contain documentation, which is accessible to tools like pydoc or Sphinx. Documentation is just a string placed right after the first line:

```
def distance(r1, r2):
    """
    :param r1: a point
    :type r1: numpy.array
    :param r2: another point
    :type r2: numpy.array
    :returns: the Euclidean distance between r1 and r2
    :rtype: float
    """
    d = r1 - r2
    return np.sqrt(np.dot(d, d))
```

In the above example, the documentation follows the conventions of Sphinx, the de-facto standard documentation generator in the Python universe.

4 Examples

4.1 Example 1

This program counts the number of lines and words in a text file.

```
lines = 0
words = 0
for line in file('some_file'):
    lines = lines + 1
    words = words + len(line.split())

print lines, " lines, ", words, " words."
```

4.2 Example 2

This program reads a file that describes a system made up of atoms. Every line contains the chemical element symbol for an atom followed by three coordinates. The program calculates the center of mass.
Note the use of a dictionary for storing the atomic masses for the chemical elements. This is a very typical example for the use of dictionaries in Python.

```
import numpy as np

masses = {'h': 1, 'c': 12, 'o': 16}
tot_mass = 0.
sum = np.array([0., 0., 0.])
for line in file('atoms'):
    element, x, y, z = line.split()
    position = np.array([float(x), float(y), float(z)])
    mass = masses[element.lower()]
    tot_mass += mass
    sum += mass*position

print 'Center of mass: ', sum/tot_mass
```