

CTAC 2006 Workshop: Python in Scientific Computing

Exercise 1 - The Python Programming Language

Ole Nielsen, Geoscience Australia and Australian National University



1 The fastest Python course in the west

This exercise aims to give a very brief introduction to selected parts of the Python programming language. During the workshop we will build and expand on this introduction as needed.

Another crash course is found at <http://www.hetland.org/python/instant-python.php>, a more comprehensive tutorial is available at <http://www.python.org/doc/current/tut/tut.html> (start at Chapter 3) and the full reference manual is available at <http://docs.python.org>.

The site www.python.org itself contains everything worth knowing about Python. You won't need to access any of the WEB resources given here to complete the demo — they are included for your reference only.

To gain access to the Python interpreter on the APAC sc computer make sure you have set the `USE_PYTHON` variable in either `.cshrc` or `.profile` as done previously in the summerschool. Please ask the tutors for help if unsure.

1.1 A first glance

The first thing to try is to invoke the Python interpreter by typing the command

```
python
```

You should see something like the following message and the python prompt (`>>>`):

```
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python is now ready to take your order!

Python can do calculations as in most other languages. For example to compute the interest earned in half a year on 5000 dollars with a rate of 4.8% per year, type:

```
>>> principal = 5000
>>> rate = 4.8
>>> years = 0.5
>>> interest = principal * (1+rate/100)**years - principal
>>> interest
118.59355682789101
```

The last command causes Python to print the result to the screen.

Python is often used to manipulate texts. Let us try print a statement from the bank. For example

```
>>> 'Initial deposit: ${d}' %(principal)
'Initial deposit: $5000'
```

Notice how the value of the decimal number `principal` is inserted in the text instead of the *placeholder* `%d`. This is an example of *string interpolation*. Other placeholders are `%f` for floating point numbers and `%s` for strings¹. A complete list of placeholders is available at

<http://www.python.org/doc/current/lib/typesseq-strings.html>. A key facet of Python's string interpolation is that it can occur anywhere that a string can.

Back to the example: Press the up-arrow key and modify the line to the following:

```
>>> s = 'Initial deposit: ${d}' %(principal)
>>> s = s + '\n'
>>> s = s + 'Interest in %d months: $%.2f.' %(years*12, interest)
>>> print s
Initial deposit: $5000
Interest in 6 months: $118.59.
```

Note that text strings can be assigned to variables and concatenated using `'+'`. The special character `'\n'` is turned into a newline when the string is printed. The

¹These codes are very similar to those used in the programming language C

first number ($\text{years} \times 12$) is inserted in the placeholder `%d` the second in `%.2f`. The `%.2f` means that the floating point number is rounded to 2 decimals.

Strings can be indexed to extract any characters or substrings. The first character in string has index 0 and the last has index -1. Substrings can be specified with the *slice notation*: two indices separated by a colon.

```
>>> s[0]
'I'
>>> s[0:3]
'Ini'
>>> s[3:5]
'ti'
>>> s[-1]
'.'
>>> s[-4:-1]
'.59'
```

Note that the index of the last character in each slice is one less than the upper bound. Note also the effect of negative indices.

All variables in Python are objects and as such they may have methods that operate on them. In addition, one may apply *introspection* to any Python object in order to gain access to their internal workings and documentation. For example, try:

```
>>> dir(s)
```

You will see a list of methods available within the string object `s`. One of the methods are `'upper'` and you can gain access to its documentation by writing

```
>>> print s.upper.__doc__
```

To invoke this method, simply type

```
>>> print s.upper()
```

The principle of introspection extends to all Python objects — including those you create yourself — and provides a powerful tool for online documentation as well as added functionalities to your programs. The site <http://www-106.ibm.com/developerworks/library/l-pyint.html> provides more information on the power of introspection..

1.2 Sequences

Python knows a number of *compound* data types, used to group together other values. One of the most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type and can even be other sequences themselves.

```
>>> a = ['spam', 'eggs', 100, 1234, [7, 'up']]
>>> a
['spam', 'eggs', 100, 1234, [7, 'up']]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated (using +) and so on.

Another very important sequence type is the *dictionary*, which is a generalised array that can be indexed by pretty much anything. Try, for example:

```
>>> german_english = {'gutentag':'hello', 'meine': 'my', 'freunde': 'friends'}
and then add some more entries in a different way
```

```
>>> german_english['alles'] = 'everything'
>>> german_english['ist'] = 'is'
>>> german_english['wichtig'] = 'important'
>>> german_english['nicht'] = 'not'
>>> german_english['relevant'] = 'relevant'
```

and then lookup a word as in

```
>>> german_english['alles']
```

Text strings are easily converted to lists as in

```
>>> text = 'gutentag meine freunde alles ist wichtig nicht alles ist relevant'
>>> L = text.split()
>>> print L
```

and backwards again (using the import string module) as in

```
>>> import string
>>> string.join(L)
```

Question: How can you find out what the strings module contains in addition to join?

To process elements of a list and return another list, Python provides the concept of *List comprehension*:

```
>>> [word.upper() for word in text.split()]
```

All the above is put together in the following (quite simplistic and arguably meaningless) translation example:

```
>>> string.join([german_english[word] for word in text.split()])
```

To exit Python simply press Ctrl-d

2 Python scripts

Writing commands directly to Python, as we have done so far, quickly becomes tedious. Python commands can therefore be collected in a file to form a *Python program* or *Python script*.

Create a file called `test.py` for example by using the xemacs editor available on the ANUSF system.

```
xemacs test.py &
```

Goto the options tab then `syntax highlighting` and tick the 'in this buffer' option. Then goto `options` again and select `save options`. This will provide colouring of the syntax in a number of programming language in addition to automatic indentation when using the TAB key.

Type or paste in the following Python program which implements a simple interaction with a conditional branch.

```
n = int(raw_input('Type an integer: '))

if (n >= 0):
    print 'The number is %d' %n
else:
    print 'You typed a negative number'
```

Notice that Python does not use `begin-end` blocks; rather Python uses a combination of a colon and indentation to define program blocks. Type it in and run it by writing `python test.py` on the command line. Functions are defined as follows. Here is an example computing the factorial of a positive integer.

```
def factorial(n):
    result = 1
    while n > 0:
```

```
    result = n*result
    n = n-1
```

```
    return result
```

Paste in the function declaration at the top of your file and modify your if statement to call it as follows:

```
n = int(raw_input('Type an integer: '))

if (n >= 0):
    print 'The number is %d' %n
    print 'The factorial n! is %d' %factorial(n)
else:
    print 'You typed a negative number'
```

This example introduced the *while* loop in the factorial function. Python also has *loops*. To print numbers from 0 to n-1 one would write

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

Notice again that the last element is n-1 as was the case with slicing of indices.

3 Numeric sequences

An important data structure in scientific computing are sequences of numbers and while Python's lists are flexible and convenient they are not suitable for large scale numerical computations due to the overhead of allowing mixed type lists and arbitrary insertions and deletions.

Python has a module `Numeric`/^{footnote}The `Numeric` module is currently being phased out and will be replaced by the more general module `numpy`. However, the modules are very similar from a user's point of view and `numpy` is still in its early days, so we stick to `Numeric` for the purpose of this exercise. `Numeric` allows a compact and efficient representation of numbers in arrays of arbitrary dimension. This is useful for applications in computational science where programs need fast operations on long vectors or matrices of floating point numbers. In addition `Numeric` comes with a library of numerical tools such as basic linear algebra functions and Fourier transforms.

`Numeric` works closely together with lists so one can easily convert from one to other as in the commandline example

```
>>> import Numeric
>>> x = Numeric.array([1, 4, 5])
>>> x
>>> x.typecode()
```

The variable `x` now points to an integer Numeric array. This type is symbolised by the character `'l'`. For other Numeric types try `>>> dir(Numeric)` and see what characters the built-in types correspond to, as in `>>> Numeric.Int`. Numeric will 'upcast' all numbers to a type that can represent them all. For example

will create an array consisting of Floating point numbers symbolised by the character `'d'`. The type can be specified explicitly as in

```
>>> import Numeric
>>> x = Numeric.array([1, 4, 5], Numeric.Float)
>>> x
>>> x.typecode()
```

or

```
>>> import Numeric
>>> x = Numeric.array([1, 3.14, 5], Numeric.Int)
>>> x
>>> x.typecode()
```

Converting a Numeric array back to a list is done with the method

```
>>> x.tolist()
```

See `dir(x)` and `dir(Numeric)` for other methods of Numeric.

One of the big advantages of Numeric arrays are vectorised operations like those available in Matlab or Fortran 90. Not only do they provide convenient notation, they also have the potential for making programs faster due to their underlying implementation.

Examples of vector operations are on our test array `x` are

```
>>> x + 5
>>> x + Numeric.ones(3)
>>> 7*x
>>> Numeric.log(x)
```

Numeric arrays can be sliced very much like lists and vector operations can be applied to slices as in

```
>>> x[:2] = 10*x[:2]
>>> x
```

Numeric arrays can have arbitrary dimensions with the number of elements in each stored in the attribute `.shape` which is a 1d tuple of length equal to the number of dimensions. For our test array `len(x.shape)` will have the value 1 and `x.shape[0]` will be 3. A 2d array can be created from a list of lists as

```
>>> A = N.array( [[2, -1, 0], [-1, 2, -1], [0, 2, -1], [1, 1, 1]] )
>>> A
>>> A.shape
```

Computing the matrix vector product Ax is carried out through the built-in function `innerproduct`:

```
>>> Numeric.innerproduct(A, x)
```

4 Plotting

Recently a plotting package has emerged which allows Python access to integrated graphics in the style of Matlab. The package is called `Matplotlib/pylab` and it integrates seamlessly with different variants of Numeric. The following example (courtesy of the `Matplotlib` manual) creates a vector of noise, adds it to a sine wave, plots the signal in one plot and its power spectrum in another.

```
from Numeric import arange, sin, exp, pi
from pylab import randn, conv, plot, psd, subplot, show

# Create signal
dt = 0.01
t = arange(0, 10, dt) # Time vector
signal = 0.1*sin(2*pi*t)

# Create noise
noise = conv(randn(len(t)), exp(-t/0.05))*dt
noise = noise[:len(t)]

# Create noisy signal and plot
noisy_signal = signal + noise
```

```
subplot(211)
plot(t, noisy_signal)
subplot(212)
psd(noisy_signal, 512, 1/dt)

show()
```

More information on Matplotlib is available at <http://matplotlib.sourceforge.net/tutorial.html>

There are many many other useful packages for Python such as VPython for plotting 3d animations of physical systems or turbo gears for creating dynamic web sites to name a few.