Computational Programming with Python

Lecture 1: First steps - A bit of everything.

Center for Mathematical Sciences, Lund University

Lecturer: Claus Führer, Alexandros Sopasakis
Why Python?

Python is . . .

- Free and open source
- It is a *scripting language* - an interpreted *not* a compiled language
- It is object oriented with modern exception handling, dynamic typing etc.
- It has plenty of libraries among others the scientific ones: linear algebra; visualisation tools: plotting, image analysis; differential equations solving; symbolic computations; statistics ; etc.
- It has possible usages: Scientific computing, scripting, web sites, text parsing, data mining, ...
Premisses

- We work with a Python version ≥ 3.6.
- We use an IPython shell
- We use the work environment Spyder
- We work (later) with the IPython notebook.
- We start our programs with the line

```
from scipy import *
```
Examples

Python may be used in *interactive* mode
(exeected in an IPython shell)

```
In [2]: x = 3
In [3]: y = 5
In [4]: print(x + y)
8
```

Note:
```
In [2]:
```

is the *prompt string* of the IPython shell. It counts the statements. In the more basic Python shell the *prompt string* is `>>>`. 
Examples: Linear Algebra

Let us solve

\[
\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot x = \begin{pmatrix} 2 \\ 1 \end{pmatrix}
\]

In [5]: from scipy.linalg import solve
In [6]: M = array([[1., 2.],
               [3., 4.]])
In [7]: V = array([2., 1.])
In [8]: x = solve(M, V)
In [9]: print(x)
[-3.  2.5]

Note: A line can be continued on the next line without any continuation symbol as long as parentheses or brackets are not closed.
More examples

Computing $e^{i\pi}$ and $2^{100}$:

In [10]: `print(exp(1j*pi))` # should return -1
(-1+1.22460635382e-16j)
In [11]: `print(2**100)`
1267650600228229401496703205376

Note: Everything following # is treated as a comment.

Computing $\zeta(x) = \sum_{k=1}^{\infty} \frac{1}{k^x}$ with $\zeta(2) = \frac{\pi^2}{6}$ gives

In [12]: `import scipy.special`
In [13]: `scipy.special.zeta(2., 1)` # x = 2
1.6449340668482266
In [14]: `pi**2/6`
1.6449340668482264
Numbers

A number may be an *integer*, a *real number* or a *complex number*. The usual operations are

- + and − addition and subtraction
- * and / multiplication and division
- ** power

```
2**(2+2)  #  16
1j**2    # -1
```
Strings

Strings are “lists” of characters, enclosed by simple or double quotes:

'valid string'
"string with double quotes"
Strings

Strings are “lists” of characters, enclosed by simple or double quotes:

'valid string'
"string with double quotes"

You may also use *triple quotes* for strings including multiple lines:

"""This is a long, long string"""
Variables
A variable is a *reference* to an object. An object may have several references. One uses the *assignment operator* \( = \) to assign a value to a variable.

Example

```python
x = [3, 4]  # a list object is created
y = x       # this object is now referenced by x and by y
del x       # we delete one of the references
del y       # all references are deleted: the object is deleted
```
A Python list is an ordered list of objects, enclosed in square brackets. One accesses elements of a list using zero-based indices inside square brackets.
List Examples

Example

L1 = [1, 2]
L1[0] # 1
L1[1] # 2
L1[2] # raises IndexError

L2 = ['a', 1, [3, 4]]
L2[0] # 'a'
L2[2][0] # 3
L2[-1] # last element: [3,4]
L2[-2] # second to last: 1
List Utilities

- The `range(n)` function can be used to fill a list with `n` elements, starting with zero:

```
list(range(5))  # [0, 1, 2, 3, 4]
```

- The `len(L)` function gives the length of a list:

```
len(['a', 1, 2, 34])  # returns 4
```

- Use `append` to append an element to a list:

```
L = ['a', 'b', 'c']
L[-1]  # 'c'
L.append('d')
L  # L is now ['a', 'b', 'c', 'd']
L[-1]  # 'd'
```
List Utilities

- `range(n)` can be used to fill list with $n$ elements, starting with zero:

  ```python
text = list(range(5))  # [0, 1, 2, 3, 4]
  ```

- `len(L)` gives the length of a list:

  ```python
  text = len(['a', 1, 2, 34])  # returns 4
  ```
List Utilities

- `range(n)` can be used to fill list with \( n \) elements, starting with zero:

  ```python
  list(range(5))  # [0, 1, 2, 3, 4]
  ```

- `len(L)` gives the length of a list:

  ```python
  len(['a', 1, 2, 34])  # returns 4
  ```

- Use `append` to append an element to a list:

  ```python
  L = ['a', 'b', 'c']
  L[-1]  # 'c'
  L.append('d')
  L  # L is now ['a', 'b', 'c', 'd']
  L[-1]  # 'd'
  ```
List Comprehension

A convenient way to build up lists is to use the *list comprehension* construct, possibly with a conditional inside.

**Definition**
The syntax of a list comprehension is

```
[<expr> for <variable> in <list>]
```

**Example**

```
L = [2, 3, 10, 1, 5]
L2 = [x*2 for x in L]  # [4, 6, 20, 2, 10]
L3 = [x*2 for x in L if 4 < x <= 10]  # [20, 10]
```
List Comprehension in Mathematics

Mathematical Notation
This is very close to the mathematical notation for sets. Compare:

\[ L_2 = \{ 2x; \ x \in L \} \]

and

\[ L_2 = [2*x \text{ for } x \text{ in } L] \]

One big difference though is that lists are ordered while sets aren’t.
Operations on Lists

- Adding two lists *concatenates* (*sammanfogar*) them:

\[
\begin{align*}
L1 & = [1, 2] \\
L2 & = [3, 4] \\
L & = L1 + L2 \quad \# \quad [1, 2, 3, 4]
\end{align*}
\]
Operations on Lists

- Adding two lists *concatenates* (sammanfogar) them:

\[
L_1 = [1, 2] \\
L_2 = [3, 4] \\
L = L_1 + L_2 \quad \# \quad [1, 2, 3, 4]
\]

- Logically, multiplying a list with an integer concatenates the list with itself several times: \( n \times L \) is equivalent to \( L + L + \cdots + L \) \( n \) times.

\[
L = [1, 2] \\
3 \times L \quad \# \quad [1, 2, 1, 2, 1, 2]
\]

(To multiply each element by \( c \), we use *arrays* instead of lists.)
for loops

for loop

A *for loop* allows to loop through a list using an *index variable*. This variable takes successively the values of the elements in the list.
for loops

for loop

A *for loop* allows to loop through a list using an *index variable*. This variable takes successively the values of the elements in the list.

Example

```python
L = [1, 2, 10]
for s in L:
    print(s * 2)  # output: 2 4 20
```
Repeating a Task

One *typical use* of the `for` loop is to repeat a certain task a fixed number of times:

```python
n = 30
for i in range(n):
    do_something  # this gets executed n times
```
Indentation

The part to be repeated in the for loop has to be properly *indented*:

```python
for elt in my_list:
    do_something()
    something_else()
    etc
print("loop finished") # outside the for block
```

Note: In contrast to other programming languages, the indentation in Python is mandatory. Many other kinds of Python structures also have to be indented, we will cover this when introducing them.
Indentation

The part to be repeated in the for loop has to be properly *indented*:

```python
for elt in my_list:
do_something()
something_else()
etc
print("loop finished") # outside the for block
```

**Note:** In contrast to other programming languages, the indentation in Python is *mandatory*.

Many other kinds of Python structures also have to be indented, we will cover this when introducing them.
Elementary Plotting

We first make the central visualization tool in Python available:

```python
from pylab import *
```

Then we generate two lists

```python
x_list=list(range(100)) # the first hundred numbers
y_list=[sqrt(x) for x in xlist]
```

And then we make a graph

```python
plot(x_list,y_list,'o')
title('My first plot')
xlabel('x')
ylabel('Square root of x')
show()
```
Elementary Plotting

My first plot

Square root of x

x

C. Führer, A. Sopasakis, Lund University, Sweden
How to run a piece of code

There are two phases in Python programming:

1. *Writing* the code
2. *Executing* (running) it

For Task 2, one needs to give the code to the Python *interpreter* which reads the code and figures out what to do with it.

Short snippets of code can be written directly in the interpreter and executed interactively. We use the interpreter *IPython* which has extra features.

For writing a larger program, one generally uses a text *editor* which can highlight code in a good way.

There are also development suites which bundle the interpreter, editor and other things into the same program.
Spyder

Example of a development environment, Spyder

```python
# -*- coding: utf-8 -*-
from __future__ import division

Created on Thu Oct 6 10:43:23 2011

@author: claus

from scipy import *
from matplotlib.pylab import *

class newton(object):
    itmax=100

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

def df(x):
    return (f(x+1.e-8)-f(x))/1.e-8
self.df=df

def __call__(self,x0,tol):
    self.x0=x0
    self.tol=tol
    x=x0

    for it in xrange(self.itmax):
        dx=-self.f(x)/self.df(x)
        xold=x
        x=xold+dx
        if abs(xold-x) < self.tol:
            self.nb_iterations=it
            return x
```

C. Führer, A. Sopasakis, Lund University, Sweden
On your computer

To install Python and additional necessary libraries for this course, follow the instructions at

http://www.maths.lth.se/na/nahelp/
IPython

IPython is an enhanced Python interpreter.
You can launch it as a stand-alone application, but it is also embedded in Spyder.

Some notes on usage:

▶ To execute the contents of a file named `myscript.py` just write `run myscript` in IPython or use the green "run" arrow in the Spyder toolbar.

▶ Use the arrow keys to visit previous commands and

▶ Use the tabulation key to auto-complete commands and names

▶ To get `help` on an object just type `?` after it and then return

▶ When you want to quit, write `exit()`
Code Examples in the Lecture

- Summing and multiplying in loops
- Summing up the numbers from 1 to n
- Finding all integers less than 100 which can be divided by 7
- Summing up all integers less than 100 which can be divided by 7
- Plotting a circle

Note the operators \% and == in Python for division with rest and for comparing two numbers. These will discussed later in more detail.