A basic introduction to Python

Ola Skavhaug[1,2] Hans Petter Langtangen[1,2]

Simula Research Laboratory[1]

Dept. of Informatics, University of Oslo[2]

February 20, 2005

Outline

List of Topics

Why Python as a scientific computing language?

- Very clean, compact, and attractive syntax
  (Python looks like pseudo code!)
- Object-oriented and generic (template) programming
- Convenient nested heterogeneous list/hash structures
- Cross-platform interface to operating system tasks
- Can build code and interfaces at run-time based on user input
- Doc strings, modules, packages to support large codes
- Support for modern software development like unit testing and mock (simulator) objects
- *Convenient and very productive programming environment!*

Can Python replace Matlab as a scientific calculator?

Probably not in many years, but:

- Matlab has a very high–level syntax, and nice plotting facilities
- Matlab is basically a high–level, interactive interface to optimized low–level LA-code
- The language itself is rather old fashion
- Matlab is expensive
- The LA-performance of Python is comparable to Matlab
- Several scientific communities has started to use Python

List of Topics

Lexical structure

- Lines are ended with line-break or a comment sign (#)
- Indentation is used to mark code blocks

```
import os # Import the os module

filename = "test.txt"
if os.path.isfile(filename):
    FH = open(filename,"r")
    for line in FH: # Loop over each line in the file
        print line.strip()
```

A simple example

- Plotting with Gnuplot:

```
import sys
from Gnuplot import Gnuplot, Data
from Numeric import arrayrange as arange, sin

try:
    L = float(sys.argv[1]) # Read first command line arg.
except:
    L = 5.0 # If not provided

dx = L/50.0
x = arange(0, L+dx/2, dx) # Efficient array
y = sin(x)                # Array operations
g = Gnuplot(persist=True) # Make the plot persistent
g.plot(Data(y, with='lines'))
```

Details

- The `import` statement:
  - Import the module `sys`:

    ```
    import sys
    ```

  - Load the class constructors Gnuplot and Data from the Gnuplot module:

    ```
    from Gnuplot import Gnuplot, Data
    ```

  - Load the function arrayrange from the module Numeric and rename it to arange

    ```
    from Numeric import arrayrange as arange
    ```

  - Load everything from the Numeric module (not generally recommended):

    ```
    from Numeric import *
    ```

More details

- Try / Except:
  - Python raises an exception if something goes wrong
  - Exceptions can be catched and processed in a controlled manner

    ```
    try:
        L = float(sys.argv[1]) # Read first command line arg.
    except:
        L = 5.0 # If not provided
    ```

  - Although being an inefficient construction, it is useful many places (E.g. to read command line arguments)
  - Do not use inside time–critical loops. Rewrite to if/else instead

Python sequences

- Python has several list structures; lists, tuples, and strings
- The elements in a sequence can be accessed by either indexing or slicing
- Lists are mutable objects, i.e., you may rebind and delete elements
- Tuples and strings are immutable; attempting to delete or rebind elements raises an exception

Lists

- Constructors

```
list1 = [] # Empty list, same as list1 = list()
list2 = [1, 2, 3, 4]
```

- Indexing

```
list2[0]  = 2 # list2 = [2, 2, 3, 4]
list2[-1] = 5 # list2 = [2, 2, 3, 5]
list2[-2] = 4 # list2 = [2, 2, 4, 5]
```

- Slicing

```
list2[:2] = [-1, -2] # list2 = [-1, -2, 4, 5]
list2[1:3]  = [0, 0] # list2 = [-1, 0, 0, 5]
list2[1:-1] = [0, 0] # Same as above
```

- Other operations

```
list1.append(3) # Append an element to a list
list3 = list1 + list2 # list3 = [3, -1, 0, 0, 5]
list3.sort()    # list3 = [-1, 0, 0, 3, 5]
del list3[-2]   # list3 = [-1, 0, 0, 5]
len(list3)      # 4
```

Tuples

- Constructors

```
idx1 = (1, 2, 3)          # Same as idx1 = 1, 2, 3
idx2 = ()                 # Same as idx2 = tuple()
idx3 = tuple([1,2,3])     # idx3 = (1, 2, 3)
```

- Same indexing and slicing as lists (access only, no assignments)
- No normal methods
- Other operations

```
idx4 = idx1[1:] + idx3[:-1] # idx4 = (2, 3, 1, 2)
len(idx4)                # 4
```

Arrays

- Two numerical extension of Python; Numeric and numarray
- Similar usage, but different APIs
- Arrays allow efficient vectorized operations
- Examples:

```
from Numeric import *
x = arange(0, 1, 0.2) # array([0.,  0.2,  0.4,  0.6,  0.8])
y = array([1, 2, 3], Float) # array([ 1.,  2.,  3.])
z = sin(x) # array([0., 0.199, 0.389, 0.565, 0.717])
```

- A major problem with `arange` is that the end point may be included or not
- Solution:

```
def sequence(min=0.0, max=None, inc=1.0, type=Float):
    if max is None:
        max = min; min=0.0
    return arange(min, max + inc/2.0, inc, type)

seq = sequence
```

Strings

- Constructors

```
filename = "test.txt"
string1 = ''            # Empty string
string2 = str(1.5)  # Create a string from a number
```

- Same indexing and slicing as lists (access only, no assignments)

```
filename[-3:] = 'tex' # Error!
filename = filename[:-3] + 'tex' # filename = 'test.tex'
```

- Strings have many methods

```
list1 = [1, 2, 3]
str1 = ", "
str2 = str1.join(map(lambda x: str(x*x), list1))
list2 = map(lambda x: sqrt(int(x)), str2.split(', '))
if list1 == list2: print "Yes" # Prints Yes
print "%s(%g) = %g" %('sin', list1[0], math.sin(list1[0]))
```

Dictionaries

- Dictionaries are containers where the elements are accessed through immutable keys (strings and tuples)
- Similar to hashes in Perl
- Examples:

```
matsparse = {} # Create empty dict
matsparse[(0,0)] = -1 # same as matsparse[0,0]
me = {'name': 'Ola Skavhaug', 'affiliation': 'SRL'}
print me['affiliation'] # 'SRL'

hash = {1:2, 2:4, 3:9}
for key in hash:
    print hash[key]
```

- Also dictionaries may be heterogeneous

```
list = []
hash = {'filename': filename, 'ofile' : open(filename, 'r')}
list.append(hash)
numbers = [float(x) for x in list[0]['ofile'].read().split()]
```

Map, reduce, and list comprehension

- Map applies a function to all elements of a sequence and returns a list of the results

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

x = seq(0,1,0.1)
y = map(square, x)
```

- List comprehension is similar to map, only a different syntax

```
x_squared = [square(elm) for elm in x]
```

- Reduce applies a function to the elements in a sequence (left to right) to reduce it to a single value

```
from operator import add
from math import sqrt
x = seq(0,1,0.1)
x_norm = sqrt(reduce(add, map(square, x)))
```

Loops

- The `for` statement:
  - Unlike C/C++ and Fortran, a for-loop in Python iterates over the items of any sequence (i.e. iterator)

```
for (a,b) in [(1,2), (3,4), (5,6)]: print a*b

for char in 'Python rules': print char

x = arange(0, 3, 0.1, Float)
for elm in x:
    elm = sin(elm) # No effect on 'x'
```

- The `while` statement:
  - Similar to C/C++ and Fortran

```
i = 0
while (i<10): i += 1
```

- `break` and `continue` may be used in loops

Functions

- Functions are created with `def`

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

- References to functions can be passed round

```
def apply(f, x):
    return f(x)
```

- Templates for free

```
print apply(square, 2.0)
4
print apply(square, seq(0, 1, 0.1))
[0. 0.01 0.04 0.09 0.16 0.25 0.36 0.49 0.64 0.81 1.]
```

- Functions can be anonymous

```
map(lambda x: x*x, arange(0, 1, 0.2, Float))
```

Two types of function arguments

- Positional arguments

```
def func1(a, b, c): return a + b + c


print func2(1) # prints 1
print func2((1, 2), "Semla", [1, 'Fika']) # prints 3
```

- Keyword arguments

```
def func3(x=1, y=2): return x + y
print func()          # Prints 3
print func(y=-1, x=1) # Prints 0
print func(2, 2)      # Prints 4
```

- "Pythonic" coding style: Always return values explicitly:

```
def alter_list(x): del x[-1]; return x

list = alter_list(list) # Same as alter_list(list)
```

More function arguments

- A function does not need to know the number of arguments

```
def func2(*a): return len(a)
func2(1,2,3,4,5) # prints 5
```

- A function does not need to know anything about the name and number of keyword arguments either

```
def func3(**kwargs): return len(a)
```

- The most general function in Python:

```
def func4(*args, **kwargs):
    print kwargs.keys()              # ['weather', 'uppsala']
    return len(args) + len(kwargs)   # 4

func4(1,2, weather='cold', uppsala='nice')
```

Classes

Python has a powerful object model

- Everything are objects in Python: Source code, classes, functions, datatypes
- "Class objects" are called instances
- Class example:

```
class QuadPoly(object):

    def __init__(self, a=1.0, b=1.0, c=1.0):
        self.a = float(a)
        self.b = float(b)
        self.c = float(c)

    def eval(self, x):
        return self.a*x*x + self.b*x + self.c

qp = QuadPoly(1, 2, 3)
print qp.eval(2) # prints 11.0
x = seq(0, 1, 0.1)
print qp.eval(x) # squares all values in x
```

Classes, continued

Operator overloading by implementing so–called special methods

- Convert to string (e.g. for printing)

  ```
  def ___str___(s):
      return 'QuadPoly(a=%f, b=%f, c=%f)' % (s.a, s.b, s.c)
  ```

- Add two QuadPolys

  ```
  def ___add___(s, o):
      return QuadPoly(s.a + o.a, s.b + o.b, s.c + o.c)
  ```

- Right add a QuadPoly to an instance

  ```
  def ___radd___(s, o):
      s.a += o.a
      s.b += o.b
      s.c += o.c
  ```

- Lots of other special methods

List of Topics

1. Motivation
2. Python basics
3. **Useful tools**
4. Python as scientific calculator
5. Resources

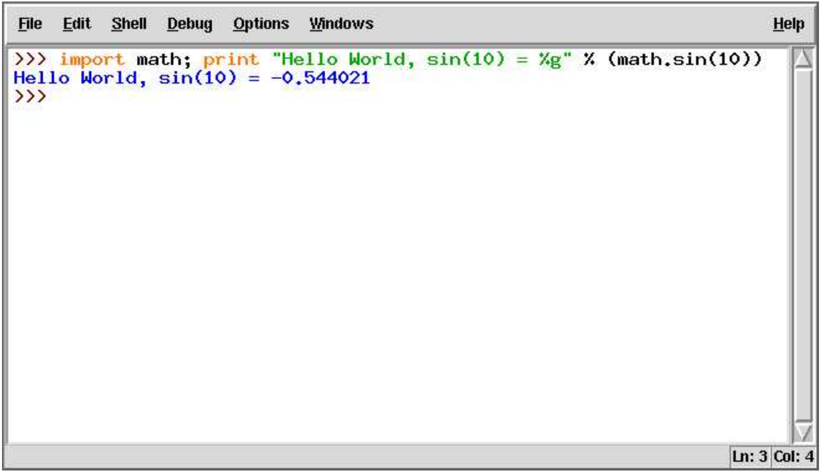
The interactive Python shell

```
# python
Python 2.3.5 (#2, Feb  9 2005, 00:38:15)
[GCC 3.3.5 (Debian 1:3.3.5-8)] on linux2
Type "help", "copyright", "credits" or "license" for ...
>>>
```

The idle shell

- InteractiveSyntax highlighting editor
- GUI frontend to pdb (python debugger)
- Box pop–up help for function arguments
- Very useful for presentations!

Idle screenshot



Profiling

The hotshot module can be used to profile Python scripts

- Consider the small script `hotshotit.py`:

```
import sys, os
script = sys.argv[1]
resfile = '.tmp.profile'
sys.path.insert(0, os.path.dirname(script)) # local modules
del sys.argv[0]  # hide the script name from sys.argv

import hotshot, hotshot.stats
prof = hotshot.Profile(resfile)
prof.run('execfile(' + `script` + ')')
p = hotshot.stats.load(resfile)
p.strip_dirs().sort_stats('time').print_stats(20)
```

Sample usage

```
    1082 function calls (728 primitive calls) in 17.890 CPU seconds

Ordered by: internal time
List reduced from 210 to 20 due to restriction <20>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     5    5.850    1.170    5.850    1.170 m.py:43(loop1)
     1    2.590    2.590    2.590    2.590 m.py:26(empty)
     5    2.510    0.502    2.510    0.502 m.py:32(myfunc2)
     5    2.490    0.498    2.490    0.498 m.py:37(init)
     1    2.190    2.190    2.190    2.190 m.py:13(run1)
     6    0.050    0.008   17.720    2.953 funcs.py:126(timer)
...
```

MayaVi and VTK

Advanced 3D plotting in Python

- Homepages: `http://mayavi.sourceforge.net` and `http://www.vtk.org`VTK is a very powerful object oriented library; it supports both structured and unstructured grids
- Using MayaVi is much easier than using VTK-Python directly
- MayaVi focuses on high–level visualization, with several pre–made modules and filters
- MayaVi comes with a GUI, and works as a stand alone visualization program as well as a Python library

List of Topics

The Trapezoidal rule

```
#!/usr/bin/env python
from scitools import *

def trapezoidal(f, a, b, n):
    '''
    Integrate f(x) from a to b using the composite Trapezoidal
    rule with n evaluation points.
    '''
    h = (b-a)/float(n-1)
    I = 0.5*f(a)
    for i in iseq(1, n-2):
        I += f(a + i*h)
    I += 0.5*f(b)
    I *= h
    return I

# verification step:
def linear(x):
    return 2 + 3*x

def integral_of_linear(a, b):
    return 2*b + (3./2)*b**2 - (2*a + (3./2)*a**2)
```

The Trapezoidal rule, continued

```
def verify():
    a = 1.5; b = 1.8; n = 4
    I = trapezoidal(linear, a, b, n)  # 4 points
    print 'verification: n=%d, I=%g error=%g' % \
          (n, I, integral_of_linear(a, b) - I)

def test(n):
    # real integral computation:
    def f(x):
        return sqrt(x)

    I = trapezoidal(f, 0, 2, n)
    exact = (2./3)*2**(3./2)
    print 'n=%d  approximation=%g  error=%g' % \
          (n, I, exact-I)

if __name__ == '__main__':
    import sys
    if sys.argv[1] == 'verify': verify()
    else:
        n = int(sys.argv[1])
        test(n)
```

A two–point boundary value problem

```
#!/usr/bin/env python

from numarray import *
from LinearAlgebra import *
import sys,Gnuplot
from scutils import sequence

def f(x):
    return (3*x+x**2)*exp(x)

try: n=int(sys.argv[1])
except: n=10
h=1./(n+1)

# fill data
x = sequence(0,1,h,'Float')
A = zeros((n+2,n+2),'Float') + identity(n+2)
A[1:-1,1:-1] += identity(n)
ind1 = range(1,n)
ind2 = range(2,n+1)
A[ind1,ind2] = A[ind2,ind1] = -1.0
b = h**2*f(x)
```

A two–point boundary value problem

```
# force boundary condition
b[0] = b[n+1] = 0
u = solve_linear_equations(A,b)

# create a simple plot
# g = Gnuplot.Gnuplot(persist=1)
g = Gnuplot.Gnuplot(persist=1)
g.title("Two point BV problem")
gdata = Gnuplot.Data(x,u,title='approx',
        with='linespoints')
g.plot(gdata)
```

List of Topics

Resources

- Alex Martelli: Python in a nutshell
- Python homepage: www.python.org
- Numerical Python homepage: http://sourceforge.net/projects/numpy