Experience with Python in a Major Computational Science Teaching Reform

Hans Petter Langtangen

Simula Research Laboratory
Dept. of Informatics, Univ. of Oslo

SIAM CSE, March 5, 2009
Teaching reform:
Numerical programming and simulation as a primary tool in science and engineering courses – from day 1

Key questions of this talk:
- Why should you consider Python for programming in computational science and engineering?
- Why is Python particularly suitable for students?
- What does Python code look like?
- What can Python be used for?
- Does Python scale to real science/engineering and high-performance computing?
Teaching reform:
Numerical programming and simulation as a primary tool in science and engineering courses – from day 1

Key questions of this talk:
- Why should you consider Python for programming in computational science and engineering?
- Why is Python particularly suitable for students?
- What does Python code look like?
- What can Python be used for?
- Does Python scale to real science/engineering and high-performance computing?
Teaching reform:
Numerical programming and simulation as a primary tool in science and engineering courses – from day 1

Key questions of this talk:
- Why should you consider Python for programming in computational science and engineering?
- Why is Python particularly suitable for students?
- What does Python code look like?
- What can Python be used for?
- Does Python scale to real science/engineering and high-performance computing?
We are implementing a major reform in science education

Where: University of Oslo

First semester:
- Classical calculus
- Numerical calculus
- Scientific computer programming

Second semester:
- Vector calculus w/numerics
- Linear algebra w/numerics
- Mechanics w/simulation

Third semester:
- More linear algebra w/numerics
- Science courses w/simulation
- Administrating numerics ("scripting")
We are implementing a major reform in science education

- Where: University of Oslo

  - First semester:
    - Classical calculus
    - Numerical calculus
    - Scientific computer programming

  - Second semester:
    - Vector calculus w/numerics
    - Linear algebra w/numerics
    - Mechanics w/simulation

  - Third semester:
We are implementing a major reform in science education

Where: University of Oslo

First semester:
- Classical calculus
- Numerical calculus
- Scientific computer programming

Second semester:
- Vector calculus w/numerics
- Linear algebra w/numerics
- Mechanics w/simulation

Third semester:
- More linear algebra w/numerics
- Science courses w/simulation
- Administrating numerics ("scripting")
We are implementing a major reform in science education

Where: University of Oslo

First semester:
- Classical calculus
- Numerical calculus
- Scientific computer programming

Second semester:
- Vector calculus w/numerics
- Linear algebra w/numerics
- Mechanics w/simulation

Third semester:
- More linear algebra w/numerics
- Science courses w/simulation
- Administrating numerics ("scripting")
We are implementing a major reform in science education

- **Where:** University of Oslo
- **First semester:**
  - Classical calculus
  - Numerical calculus
  - Scientific computer programming
- **Second semester:**
  - Vector calculus w/ numerics
  - Linear algebra w/ numerics
  - Mechanics w/ simulation
- **Third semester:**
  - More linear algebra w/ numerics
  - Science courses w/ simulation
  - Administrating numerics (”scripting”)

We are implementing a major reform in science education

- **Place:** University of Oslo

**First semester:**
- Classical calculus
- Numerical calculus
- **Scientific computer programming**

**Second semester:**
- Vector calculus w/numerics
- Linear algebra w/numerics
- Mechanics w/simulation

**Third semester:**
- More linear algebra w/numerics
- Science courses w/simulation
- Administrating numerics ("scripting")
Demands to the introductory programming course

Contents:

- Programming with formulas, loops, lists, functions
- Curve plotting and array computing
- Computing with sequences and sound
- Discrete calculus computing
- File handling, hash tables
- Class programming
- Random numbers and Monte Carlo simulation
- ODE computing
- Object-oriented programming (class hierarchies)
What shall the students learn? A look at the final project

Numerical solution of

\[ m\ddot{u} + f(\dot{u}) + s(u) = F(t), \quad u(0) = U_0, \quad \dot{u}(0) = V_0 \]

For example, pendulum with air resistance:

\[ m\ddot{u} + \frac{1}{2}C_D \rho A|\dot{u}|\dot{u} + mg \sin(u) = F(t), \quad u(0) = U_0, \quad \dot{u}(0) = 0 \]

- Matlab-style implementation
- Visualization: \( u(t), \dot{u}(t), \ddot{u} \) vs. \( u \)
- Class-style implementation (class Problem, Solver, Viz)
- Class hierarchy for choices of \( f, s \) and \( F \)
  (e.g., \( f(\dot{u}) = 0 \), or \( \beta \dot{u} \), or \( \frac{1}{2}C_D \rho A|\dot{u}|\dot{u} \))
First programming course: what should be the language?

Matlab-style program

Java/C++ systems
Python offers a smooth transition from Matlab-style programming to the style of Java/C++
Python is used for a variety of applications:

- Unix shell scripts
- Text/file processing
- GUI
- Web services
- Large software systems
- Teaching: 1st language
- Matlab/IDL alternative
- New life (interfaces) to F77/C code
Python is a convenient programming environment

def myfunc(x, y, t):
    return sinh(x)*cosh(y)*exp(-0.15*t)

t = 0
while t < T:
    A, b = matrixfactory(grid, u, myfunc)
    P, status = ML.preconditioner(A)
    x = linear_solver(A, b, M)
    u.set_new_values(x)
    vtk.visualize(u, t)
    netcdf.store(u, t); pickle.dump(u)
    GUI.update(t)
    t += dt

Very clean syntax, high-level statements, “executable pseudo code”
Python is a convenient programming environment.

Variables can hold objects of any type.

```python
def myfunc(x, y, t):
    return sinh(x) * cosh(y) * exp(-0.15 * t)

T = 0
while t < T:
    A, b = matrixfactory(grid, u, myfunc)
    P, status = ML.preconditioner(A)
    x = linear_solver(A, b, M)
    u.set_new_values(x)
    vtk.visualize(u, t)
    netcdf.store(u, t); pickle.dump(u)
    GUI.update(t)
    t += dt
```
Python is a convenient programming environment

def myfunc(x, y, t):
    return sinh(x)*cosh(y)*exp(-0.15*t)

t = 0
while t < T:
    A, b = matrixfactory(grid, u, myfunc)
    P, status = ML.preconditioner(A)
    x = linear_solver(A, b, M)
    u.set_new_values(x)
    vtk.visualize(u, t)
    netcdf.store(u, t); pickle.dump(u)
    GUI.update(t)
    t += dt
def myfunc(x, y, t):
    return sinh(x)*cosh(y)*exp(-0.15*t)

t = 0
while t < T:
    A, b = matrixfactory(grid, u, myfunc)
    P, status = ML.preconditioner(A)
    x = linear_solver(A, b, M)
    u.set_new_values(x)
    vtk.visualize(u, t)
    netcdf.store(u, t); pickle.dump(u)
    GUI.update(t)
    t += dt

subroutine generate(A, n, nz, q)
integer n, nz, i
real*8 A(0:n-1, 0:nz-1, q
do i =1,n
  A(i,q) = A(i-1,q) + f(i,q)
Python is a convenient programming environment

def myfunc(x, y, t):
    return sinh(x)*cosh(y)*exp(-0.15*t)

    t = 0
    while t < T:
        A, b = matrixfactory(grid, u, myfunc)
        P, status = ML.preconditioner(A)
        x = linear_solver(A, b, M)
        u.set_new_values(x)
        vtk.visualize(u, t)
        netcdf.store(u, t); pickle.dump(u)
        GUI.update(t)
        t += dt

subroutine generate(A, n, nz, q)
    integer n, nz, i
    real*8 A(0:n-1, 0:nz-1, q
    do i =1,n
        A(i,q) = A(i-1,q) + f(i,q)
Python is a convenient programming environment

```python
def myfunc(x, y, t):
    return sinh(x)*cosh(y)*exp(-0.15*t)

boost = 0
while t < T:
    A, b = matrixfactory(grid, u, myfunc)
    P, status = ML preconditioner(A)
    x = linear_solver(A, b, M)
    u.set_new_values(x)
    vtk.visualize(u, t)
    netcdf.store(u, t); pickle.dump(u)
    GUI.update(t)
    t += dt
```

GUI toolkits, e.g., Tk, Gtk, Qt, wxWidgets

subroutine generate(A, n, nz, q)
integer n, nz, i
real*8 A(0:n-1, 0:nz-1), q
do i =1,n
    A(i,q) = A(i-1,q) + f(i,q)
```
Python is a convenient programming environment

```python
def myfunc(x, y, t):
    return sinh(x)*cosh(y)*exp(-0.15*t)

t = 0
while t < T:
    A, b = matrixfactory(grid, u, myfunc)
    P, status = ML.preconditioner(A)
    x = linear_solver(A, b, M)
    u.set_new_values(x)
    vtk.visualize(u, t)
    netcdf.store(u, t); pickle.dump(u)
    GUI.update(t)
    t += dt
```

Easy interfacing to Fortran, C, C++ codes; great simplification of interfaces is possible

```fortran```
subroutine generate(A, n, nz, q)
    integer n, nz, i
    real*8 A(0:n-1, 0:nz-1), q
    do i =1,n
        A(i,q) = A(i-1,q) + f(i,q)
```
Python is a convenient programming environment

```python
def myfunc(x, y, t):
    return sinh(x)*cosh(y)*exp(-0.15*t)
```

- Matlab-ish arrays and array computing
- Flexible data structures (hash, list, class)
- Great software engineering support (packages, unit tests, documentation)
- Python scales from small sessions/scripts to large software systems
- Rich standard library
- Wide collection of third-party modules
- Cross-platform operating system interface
- Support of all major programming styles
- Overloaded operators
- I/O tools (pickle, shelf, netCDF, HDF5)
- Regular expressions for text processing
- Run-time code generation
- Free, open source
Example: find the roots of a quadratic equation

Solve $ax^2 + bx + c = 0$

- Python (as Matlab) has transparent complex/real arithmetics
  
  ```python
  from numpy.lib.scimath import sqrt
  
  def roots(a, b, c):
      q = sqrt(b**2 - 4*a*c)  # q is real or complex
      return (-b + q)/(2*a), (-b - q)/(2*a)
  
  x1, x2 = roots(-1, 10, 3.1)
  ```

- Java: use class Complex, no operator overloading; Complex and double require very different implementations

- C: in principle as Java

- C++:
  ```cpp
  template function <class T>
  void roots(T a, T b, T c,
             std::complex<double>& r1, std::complex<double>& r2) { 
      std::complex<double> q = sqrt(b**2 - 4*a*c); ...
  }
  ```

Problem: r1, r2 are always complex, even when roots are real
Example: find the roots of a quadratic equation

Solve \( ax^2 + bx + c = 0 \)

- Python (as Matlab) has transparent complex/real arithmetics
  ```python
  from numpy.lib.scimath import sqrt

  def roots(a, b, c):
      q = sqrt(b**2 - 4*a*c) # q is real or complex
      return (-b + q)/(2*a), (-b - q)/(2*a)
  
  x1, x2 = roots(-1, 10, 3.1)
  ```

- Java: use class Complex, no operator overloading; Complex and double require very different implementations

- C: in principle as Java

- C++:
  ```cpp
template function <class T>
void roots(T a, T b, T c,
    std::complex<double>& r1, std::complex<double>& r2) {
    std::complex<double> q = sqrt(b**2 - 4*a*c); ...
}
```

Problem: \( r1, r2 \) are always complex, even when roots are real
Example: find the roots of a quadratic equation

Solve $ax^2 + bx + c = 0$

- **Python (as Matlab) has transparent complex/real arithmetics**
  
  ```python
  from numpy.lib.scimath import sqrt
  
  def roots(a, b, c):
      q = sqrt(b**2 - 4*a*c)  # q is real or complex
      return (-b + q)/(2*a), (-b - q)/(2*a)
  
  x1, x2 = roots(-1, 10, 3.1)
  ```

- **Java: use class Complex, no operator overloading; Complex and double require very different implementations**

  - C: in principle as Java
  - C++:
    ```cpp
    template function <class T>
    void roots(T a, T b, T c,
               std::complex<double>& r1, std::complex<double>& r2) {
        std::complex<double> q = sqrt(b**2 - 4*a*c); ... 
    }
    ```

  Problem: $r1, r2$ are always complex, even when roots are real
Example: find the roots of a quadratic equation

Solve $ax^2 + bx + c = 0$

- Python (as Matlab) has transparent complex/real arithmetics
  
  ```python
  from numpy.lib.scimath import sqrt
  
  def roots(a, b, c):
      q = sqrt(b**2 - 4*a*c)  # q is real or complex
      return (-b + q)/(2*a), (-b - q)/(2*a)
  
  x1, x2 = roots(-1, 10, 3.1)
  ```

- Java: use class Complex, no operator overloading; Complex and double require very different implementations

- C: in principle as Java

- C++:
  
  ```cpp
  template function <class T>
  void roots(T a, T b, T c,
              std::complex<double>& r1, std::complex<double>& r2) {
      std::complex<double> q = sqrt(b**2 - 4*a*c); ... 
  }
  ```

  Problem: r1, r2 are always complex, even when roots are real
Example: find the roots of a quadratic equation

\[ ax^2 + bx + c = 0 \]

- Python (as Matlab) has transparent complex/real arithmetics
  ```python
  from numpy.lib.scimath import sqrt

  def roots(a, b, c):
      q = sqrt(b**2 - 4*a*c)  # q is real or complex
      return (-b + q)/(2*a), (-b - q)/(2*a)
  
  x1, x2 = roots(-1, 10, 3.1)
  ```

- Java: use class Complex, no operator overloading; Complex and double require very different implementations

- C: in principle as Java

- C++:
  ```cpp
template function <class T>
void roots(T a, T b, T c,
    std::complex<double>& r1, std::complex<double>& r2) {
    std::complex<double> q = sqrt(b**2 - 4*a*c); ...
}
```

Problem: r1, r2 are always complex, even when roots are real
Easy: 1) several functions in a file, 2) math functions as arguments to functions

Python: compute $f'(x) \approx \left( f(x + h) - f(x) \right) / h$

```python
def differentiate(f, x, h=1.0E-9):
    return (f(x+h) - f(x))/h

def g(x):
    return x**2*sin(2*x+1)
dg = differentiate(g, x=0.1)
```

Matlab version:

```matlab
% file differentiate.m:
function df = differentiate(f, x, h)
f = fcnchk(f)
df = (feval(f, x+h) - feval(f, x))/h        % old syntax
df = (f(x+h) - f(x))/h                     % new syntax

% file g.m:
function y = g(x)
y = t*t*sin(2*t+1)

% file main.m:
result = differentiate(@g, 0.1, 1E-9)
disp(result)
```
interface Func {
   // base class for functions f(x)
   public double eval (double x);
}

class ComputeDerivative {
   public static double differentiate
      (Func f, double x, double h=1.0E-9)
   {
      return (f.eval(x+h) - f.eval(x))/h;
   }
}

class G implements Func {
   public double eval (double x)
   {
      return Math.pow(x, 2)*Math.sin(2*x+1);
   }
}

class Demo {
   public static void main (String argv[])
   {
      G g = new G();
      result = ComputeDerivative.differentiate(g, 0.1);
      System.out.println(result);
   }
}
With a smart class we can have "automatic differentiation"

**Goal:** given $f(x)$, create automatically $g(x) \approx f'(x)$

```python
def f(x):
    return x**5

g = Derivative(f)
# now g(x) behaves as an ordinary function 5*x**4 (approx)
value = g(0.1)
```

**Implementation in terms of a class:**

```python
class Derivative:
    def __init__(self, f, h=1E-9):  # constructor
        self.f = f
        self.h = float(h)

    def __call__(self, x):  # operator ()
        f, h = self.f, self.h  # make short forms
        return (f(x+h) - f(x-h))/(2*h)
```
Back to the final course work

Numerical solution of

\[ m \ddot{u} + f(\dot{u}) + s(u) = F(t), \quad u(0) = U_0, \quad \dot{u}(0) = V_0 \]

For example, pendulum with air resistance:

\[ m \ddot{u} + \frac{1}{2} C_D \rho A |\dot{u}| \dot{u} + mg \sin(u) = F(t), \quad u(0) = U_0, \quad \dot{u}(0) = 0 \]
from ODESolver import *
from scitools.all import *  # arrays + plotting

def rhs(state_vector, t):
    u, dudt = state_vector
    return [u,
            \(-(f(dudt) + s(u)) + F(t))/m\]

dt = pi/10  # time step
for Method in ForwardEuler, RK2, Midpoint, RK4:
    method = Method(rhs, dt)
    method.set_initial_condition([[0.3, 0]])
    solution, t = method.solve(T=4*pi)
    u, dudt = solution[:,0], solution[:,1]
    plot(t, u)
    hold('on')
    label(Method.__name__)

# analytical solution:
    u_exact = 0.3*cos(t)
    plot(t, u_exact)
    label('exact')
title('Comparison of methods')
Add-on packages for getting Matlab-like functionality

- **numpy**: arrays and array computing
- **scitools.easyviz**: Matlab-like syntax for plotting – unified interface to graphics packages (Matplotlib, Gnuplot, Matlab, Vtk, OpenDX, Grace, ...)
- **scipy**: special math functions, most of LAPACK, Netlib packages for integration, ODEs, etc.
- **sympy**: symbolic math

How is the student feedback?

- Hard work
- Very interesting and motivating because it seems relevant
- Gives better understanding of mathematics
- Exploration of models is fun
- Not well suited for weak students
- Some have installation problems
Installing Python and all extra software might be a nightmare

- Platform jungle: Mac OS X, Windows XP/Vista, Linux RedHat/Ubuntu/Suse, ...
- Basic Python is easy
- NumPy, SciPy, Gnuplot, VTK, ATLAS, gcc, Tcl/Tk, libpng, libz, Qt, wxWidgets, ...
- Sage and Enthought editions are great!
- Users want just one package
- Virtual Ubuntu? One-line install...
- Teachers struggle more than students
Can Python be used for real scientific problems?

**Typical use of Python in the HPC world:**
- Glue existing programs/libraries
- Write new applications

**What about efficiency?**
- Non-numeric code is often fast
- Loops over long arrays run slowly
- Loops can be migrated to compiled code (F77, C, C++)
HPC = Python + compiled loops

\[ \frac{\partial^2 u}{\partial t^2} = \nabla \cdot (k \nabla u) \]

Wave equation with finite differences in 3D, harmonic mean of \( k \)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>stand-alone Fortran 77 program</td>
<td>1.0</td>
</tr>
<tr>
<td>Py + F77 loops (handcoded)</td>
<td>1.0</td>
</tr>
<tr>
<td>Py + C loops (handcoded)</td>
<td>1.0</td>
</tr>
<tr>
<td>Py + F77 loops via F2PY</td>
<td>1.1</td>
</tr>
<tr>
<td>Py + C loops via Weave</td>
<td>1.1</td>
</tr>
<tr>
<td>Py + C++ loops via Instant</td>
<td>1.2</td>
</tr>
<tr>
<td>Py + Cython</td>
<td>1.4</td>
</tr>
<tr>
<td>Py + vectorization via slices</td>
<td>2.8</td>
</tr>
<tr>
<td>Psyco-accelerated plain loops, numpy</td>
<td>130</td>
</tr>
<tr>
<td>Py w/plain loops, numpy arrays</td>
<td>150</td>
</tr>
</tbody>
</table>
def calculate_u(dt, dx, dy, u, um, up, k):
    n, m = u.shape
    c_code=r'''
    int i, j;
    double dt_ = dt;
    for (i=1; i<n-1; i++)
        for (j=1; j<m-1; j++)
            up[i*n+j] = 2*u[i*n+j] - um[i*n+j] + pow((dt_/dx),2)*((0.5*
    ''',
    err = weave.inline(c_code, ['n', 'm', 'dt', 'dx', 'dy',
                                'u', 'um', 'up', 'k'])
    return up
def timeloop(t, t_stop, dt, dx, dy, u, um, up, k):
    while t <= t_stop:
        t += dt
        up = calculate_u(dt, dx, dy, u, um, up, k)
        um[:] = u
        u[:] = up
import numpy as np
cimport numpy as np
cimport cython
DTYPE = np.float
ctypedef np.float_t DTYPE_t
@cython.boundscheck(False)
def calculate_u(float dt, float dx, float dy,
np.ndarray[DTYPE_t, ndim=2, negative_indices=False] u,
np.ndarray[DTYPE_t, ndim=2, negative_indices=False] um,
np.ndarray[DTYPE_t, ndim=2, negative_indices=False] up,
np.ndarray[DTYPE_t, ndim=2, negative_indices=False] k):
cdef int n = u.shape[0]-1
cdef int m = u.shape[1]-1
cdef int i, j, start = 1

for i in xrange(start, n):
    for j in xrange(start, m):
        up[i,j] = 2*u[i,j] - um[i,j] + (dt/dx)**2*((0.5*(k[i+1,j]
return up
calculate_u_code = ""
    subroutine calculate_u(dt, dx, dy, u, um, up, k, n, m)
        integer n, m
        real*8 u(0:n, 0:m), um(0:n, 0:m)
        real*8 up(0:n, 0:m), k(0:n, 0:m)
        real*8 dt, dx, dy
        Cf2py intent(in) u, up, k
        Cf2py intent(out) up
        integer i, j
        do j = 1, m-1
            do i = 1, n-1
                up(i,j) = 2*u(i,j) - um(i,j) +
                & (dt/dx)*(dt/dx)*
                & ((0.5*(k(i+1,j) + k(i,j))*(u(i+1,j) - u(i,j)) -
                & 0.5*(k(i,j) + k(i-1,j))*(u(i,j) - u(i-1,j))) +
                & (dt/dy)*(dt/dy)*
                & ((0.5*(k(i,j+1) + k(i,j))*(u(i,j+1) - u(i,j)) -
                & 0.5*(k(i,j) + k(i,j-1))*(u(i,j) - u(i,j-1))))
            end do
        end do
        return
    end
    """
try:
    from calculate_uf import calculate_u
except:
    f2py.compile(calculate_u_code, modulename='calculate_uf',
                 verbose=0, extra_args=compiler)
    from calculate_uf import calculate_u

def timeloop(t, t_stop, dt, dx, dy, u, um, up, k):
    while t <= t_stop:
        t += dt
        up = calculate_u(dt, dx, dy, u, um, up, k)
        um[:] = u
        u[:] = up
"""
from dolfin import * # www.fenics.org package
mesh = UnitSquare(32, 32)
V = FunctionSpace(mesh, "CG")

v = TestFunction(V)
U = TrialFunction(V)
f = Function(V, "sin(x[0])*cos(x[1])")

a = dot(grad(v), grad(U))*dx
L = v*f*dx

A = assemble(a); b = assemble(L); u_h = Function(V)
solve(A, u_h.vector(), b)
plot(u_h)
mpi4py can parallelize high-level Python codes

High-level interface to MPI:

- Arbitrary Python objects can be sent via MPI
- Efficient treatment of NumPy arrays
- Can parallelize "black-box" legacy codes via domain decomposition
- Older alternatives: pypar, PyMPI
What have you learned?

Students *can* learn math, numerics & programming and use it to explore science applications – from day 1!

Python...

- is an easy-to-learn, fun, clean, flexible, popular and very productive programming language
- is excellent as a 1st programming language – in particular for teaching CSE
- scales to large simulation codes
- can be used to glue various codes
- can be used to write HPC codes
- needs better installation & doc