Visions of Python in Scientific Computing

Ola Skavhaug[1,2]

Simula Research Laboratory[1]

Dept. of Informatics, University of Oslo[2]

March 7, 2005

Outline

1. Education
2. Code Reuse
3. Application Development

Scripting languages are popular

- Syntax is compact and clean; almost like pseudo–code
- Fast code development; interpreted languages do not need compilation
- Extensive general purpose libraries
- *Scripting languages constitute productive programming environments*
- Popular scripting languages: Perl, Ruby, Tcl, Python (and bash)
- Major drawback; numerical efficiency

List of Topics

1. Education
2. Code Reuse
3. Application Development

Education I

- Students want it:
  - Simple language with few pit–falls
  - Easy to learn the language by exploring the Python shells
  - Similar to other, modern languages (i.e. Java)
- Industry needs students with scripting knowledge:
  - A major challenge in industry is to automate tasks
  - This often involves gluing applications together using a scripting language

Education II

- A major problem with low–level languages is the distance between the mathematics (algorithms) and implemented code
- Using a high–level scripting language in mathematics education closes this gap
- The result may be an increased focus on algorithms, and less focus on implementation hassle
- The poor numerical efficiency is probably not important in educational settings

Example

```
def trapezoidal(f, a=0.0, b=1.0, n=10):
    '''
    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
```

List of Topics

1. Education
2. Code Reuse
3. Application Development

What about the existing code?

- Over the years, computational science groups tend to develop a huge base of legacy code
- Typical legacy code characteristics:
  - Stable, high–quality, efficient
  - Difficult to learn, use, and change
- A problem is that the demands on a given code are increasing; parallelization, GUI front-end, data conversion, advanced IO, etc.
- Quality code should be re–used
- Equipping legacy code libraries with scripting interfaces may solve the problem. The good news is that this is easy

Wrapper code tools

- C/C++:
  - SWIG – `http://www.swig.org`
    Mature, general purpose. Choosing a general solution to an efficient one. Excellent documentation
  - Sip – `http://www.riverbankcomputing.co.uk/sip/`
    Very special purpose (make a Python interface to QT), one developer. Quite efficient, almost no documentation
  - Boost – `http://www.boost.org`
    Based on template meta programming. Good documentation. Difficult to get started
  - Babel – `http://www.llnl.gov/CASC/components/babel.html`
- Fortran:
  - F2py – `http://cens.ioc.ee/projects/f2py2e`
    Tightly integrated with NumPy. Good documentation. Automatic Python callback support.

SWIG Example

Consider the following C function (`fact.c`):

```
int fact(int i) {
  if (i <= 1) return 1;
  else return i*fact(i-1);
```

A corresponding interface file (`fact.i`) may read:

```
%module fact // fact is the module name
%{
/* Put headers and other declarations here */
#include <fact.h>
%}
/* The interface definition (e.g. function signatures) */
int fact(int i);
```

The wrapper code (`fact_wrap.c`) is generated by running:

```
swig -python  fact.i
```

SWIG Example, continued

At last, the source code and the generated wrapper code must be compiled and linked:

```
> gcc -c -fpic  fact_wrap.c fact.c -I. -DHAVE_CONFIG_H  \
  -I/local/include/python2.3 -I/local/lib/python2.3/config
> gcc -shared fact.o  fact_wrap.o  -lswigpy \
  -L/local/lib/ -o _fact.so
```

In Python:

```
>>> from fact import fact
>>> fact(4)
24
```

Advanced SWIG

- A Python extension module should look and feel like native Python
- SWIG provides so–called directives to control the wrapper code generation
- Python *special methods* can often be implemented by renaming existing methods

```
%rename(__add__) add;
```

- Types can be mapped using typemaps

```
/* Convert from Python --> C */
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}

/* Convert from C --> Python */
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

Benefits of interfacing

- Sequential code may be parallelized at the scripting level, using e.g. PyMPI (http://pympi.sourceforge.net/) or Scientific.BSP (http://starship.python.net/~hinsen/ScientificPython/)
- Old libraries can be given modern, object–oriented interfaces
- Example: SciPy (http://www.scipy.org/) uses ALTAS/BLAS from netlib

List of Topics

1. Education
2. Code Reuse
3. Application Development

Applications can be developed in a scripting language

- A recent trend in scientific scripting: Design applications in a high–level scripting environment, and migrate hotspots and bottlenecks to compiled code
- Benefits:
  - Simple mapping between the Python code and the underlying mathematical problem
  - Advanced functionality (file handling and IO, GUI, initialization etc.) is easy to incorporate in an application
  - By designing the user interface first, time–critical parts of an application are easy to spot and speed up as Python extension modules

Challenges

- There should be a standardized set of data types for vectors and matrices, both scalar and distributed
- Today, even NumPy is split in two (`Numeric` vs. `numarray`)
- Installing Python extensions can be extremely difficult
- The look and feel of the Python shell as a scientific calculator must improve (i.e. better plotting and more functionality)