Extending Python to Fortran/C/C++
exploring the options

Yogesh Wadadekar

NCRA-TIFR

November 2014
Three main reasons to extend Python

1. speed.
2. to leverage existing libraries that are not already wrapped.
3. to allow Python programs to access devices at absolute memory addresses, using C functions as an intermediary.

If neither of these reasons apply, you can happily code only in Python.
Questions to answer before trying to extend Python

- is the compiled language code you need small enough to recode in Python. e.g. a Numerical recipes routine that requires perhaps 3-4 other routines should be easily re-writable in Python.

  - if the code is large, check whether it already has a Python interface. e.g. LAPACK - the linear algebra package.
  - if the code in question is in octave, idl, matlab, mathematica and you have a licensed copy of these software, it will be easier to use the interfaces to those languages.

  - Do you really need to pass arrays or are disk files a good way of exchanging data?

  - It is more efficient to wrap a set of related subroutines rather than just one (or a few). Can you collaborate to do this? Ask on appropriate mailing list.
Questions to answer before trying to extend Python

- is the compiled language code you need small enough to recode in Python. e.g. a Numerical recipes routine that requires perhaps 3-4 other routines should be easily rewritable in Python.
- if the code is large, check whether it already has a Python interface. e.g. LAPACK - the linear algebra package.
Questions to answer before trying to extend Python

- is the compiled language code you need small enough to recode in Python. e.g. a Numerical recipes routine that requires perhaps 3-4 other routines should be easily rewritable in Python.

- if the code is large, check whether it already has a Python interface. e.g. LAPACK - the linear algebra package.

- if the code in question is in octave, idl, matlab, mathematica and you have a licensed copy of these software, it will be easier to use the interfaces to those languages.
Questions to answer before trying to extend Python

- is the compiled language code you need small enough to recode in Python. e.g. a Numerical recipes routine that requires perhaps 3-4 other routines should be easily rewritable in Python.
- if the code is large, check whether it already has a Python interface. e.g. LAPACK - the linear algebra package.
- if the code in question is in octave, idl, matlab, mathematica and you have a licensed copy of these software, it will be easier to use the interfaces to those languages.
- Do you really need to pass arrays or are disk files a good way of exchanging data?
Questions to answer before trying to extend Python

- is the compiled language code you need small enough to recode in Python. e.g. a Numerical recipes routine that requires perhaps 3-4 other routines should be easily rewritable in Python.
- if the code is large, check whether it already has a Python interface. e.g. LAPACK - the linear algebra package.
- if the code in question is in octave, idl, matlab, mathematica and you have a licensed copy of these software, it will be easier to use the interfaces to those languages.
- Do you really need to pass arrays or are disk files a good way of exchanging data?
- It is more efficient to wrap a set of related subroutines rather than just one (or a few). Can you collaborate to do this? Ask on appropriate mailing list.
Numpy C extensions

http://www.scipy.org/Cookbook/C_Extensions/NumPy_arrays
SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is used with different types of languages such as Perl, PHP, Python, Tcl, Ruby, C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), Java, Lua, Modula-3, OCAML, Octave and R. Also several interpreted and compiled Scheme implementations (Guile, MzScheme, Chicken) are supported. SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software.
example.c - the C functions to be wrapped
Now, in order to add these files to your favorite language, you need to write an “interface file” which is the input to SWIG. An interface file for the C functions we want to wrap is needed.
Building the module

```
$ swig -python example.i
$ gcc -fPIC -c example.c example_wrap.c
     -I/usr/include/python2.7
$ ld -shared example.o example_wrap.o -o _example.so
```
Dynamic typing gives way to static typing

You should use an assert before the function call to the external function.
Robert Lupton wrapped the entire SM program in a few days with SWIG.
There are three basic ways to use weave. 1. The `weave.inline()` function executes C code directly within Python, and 2. `weave.blitz()` translates Python NumPy expressions to C++ for fast execution. `blitz()` was the original reason weave was built. 3. For those interested in building extension libraries, the `ext_tools` module provides classes for building extension modules within Python.
>>> import scipy.weave as weave
>>> a = 1
>>> weave.inline('printf("%d
n",a);',[a])
F2PY - Fortran to Python interface generator

- autogenerates interface files to allow Python to call a Fortran subroutine.
Recent versions of numpy already include f2py. So, if you installed numpy there is nothing more to do.
The Hello example

C File hello.f
subroutine foo (a)
integer a
print*, "Hello from Fortran!"
print*, "a=", a
end

Of course, there may be multiple subroutines in hello.f, all of which will get wrapped in one go.
Build the module

$ f2py -c -m hello hello.f
__doc__ are autocreated.
A more complicated example - fib3.f

Calculate the first $n$ Fibonacci numbers.
If you can’t edit the Fortran code, you can work with signature files which are like the interface files in SWIG.
Features of f2py

- F2PY automatically generates `__doc__` strings (and optionally LaTeX documentation) for extension modules.
- F2PY generated functions accept arbitrary (but sensible) Python objects as arguments. The F2PY interface automatically takes care of type-casting.
- Most Fortran 90 features also work.

Pyfort, a competitor to f2py is not as feature rich
Attributes and statements of f2py

intent([ in | inout | out | hide | in,out | inout,out | c | copy | cache | callback | inplace | aux ])
dimension(<dimspec>)
common, parameter
allocatable
optional, required, external
depend([<names>])
check([<C-booleanexpr>])
note(<LaTeX text>)
usercode, callstatement, callprotoargument, threadsafe, fortranname
pymethoddef
entry
Other options for extending Python

- SIP: lighter version of SWIG. Only works with C/C++. Originally written for PyQt
- ctypes: no interface code needed, can directly call compiled functions in a binary library file. Some features work only on Windows.
- Boost.python: good for wrapping entire libraries in C or C++
- Pyrex: is a Python like language designed for writing Python extension modules. Almost any Python code is valid Pyrex code.
- scipy.weave
Java: Jython is an implementation of Python for the JVM. Jython takes the Python programming language syntax and enables it to run on the Java platform. This allows seamless integration with the use of Java libraries and other Java-based applications. But CPython extensions don’t work.

C# - IronPython is an implementation of the Python programming language running under .NET and Silverlight
My recommendations

- see if numpy C extensions will do the job for you.
- if you use Fortran (any flavour), use f2py. If you need to write fresh code (for speedy execution), write in Fortran and wrap with f2py.
- For C/C++ use SWIG or may SIP
- For Java - use Jython, for C# use IronPython, no real options here
- For all other languages, SWIG is the richest option
- and please don’t forget regression tests.
Happy Wrapping!