Combining Python with Fortran, C and C++

This section follows closely the approach of Langtangen: Python Scripting for Computational Science and Engineering, chapter 5.

When is integration of Python with Fortran, C, or C++ of interest?

- migration of slow parts of a Python code to Fortran or C/C++
- access to existing numerical code (from Python)

Typically: user interfaces, i/O, report generation and management is in Python, while the computationally intensive functions are in Fortran/C/C++

A source of troubles: Fortran/C/C++ has strong typing rules, while in Python variables are typeless...

Needs writing wrapper code: each function need a Python wrapper. Automatic generation: SWIG for C/C++, f2py for Fortran

Scientific Hello World Examples

Pure Python implementation:

```
#!/usr/bin/env python
"""Pure Python Scientific Hello World module."""
import math, sys

def hw1(r1, r2):
    s = math.sin(r1 + r2)
    return s

def hw2(r1, r2):
    s = math.sin(r1 + r2)
    print 'Hello, World! sin(%g+%g)=%g' % (r1,r2,s)
```

with application script:

```
#!/usr/bin/env python
"""Scientific Hello World script using the module hw."""
import sys
from hw import hw1, hw2
try:
    r1 = float(sys.argv[1]);  r2 = float(sys.argv[2])
except IndexError:
    print 'Usage:', sys.argv[0], 'r1 r2'; sys.exit(1)
print 'hw1, result:', hw1(r1, r2)
print 'hw2, result: ',
hw2(r1, r2)
```

We intend to migrate the functions hw1 and hw2 in the hw module to Fortran/C/C++. We eill also onvolve a third function hw3, which is a version

Table Of Contents

Combining Python with Fortran, C and C++

- Scientific Hello World Examples
- Combining Python and Fortran
- Building an Extension Module with Distutils

Previous topic

Input / Output of numerical data

Next topic

Integrated development environment

This Page

Show Source

of hw1 where s is an output argument, in call by reference style, and not a return variable. A pure implementation of hw3 has no meaning.

Combining Python and Fortran

```
      program hwtest
      real*8 r1, r2 ,s
      r1 = 1.0
      r2 = 0.0
      s = hw1(r1, r2)
      write(*,*) 'hw1, result:',s
      write(*,*) 'hw2, result:'
      call hw2(r1, r2)
      call hw3(r1, r2, s)
      write(*,*) 'hw3, result:', s
      end

      real*8 function hw1(r1, r2)
      real*8 r1, r2
      hw1 = sin(r1 + r2)
      return
      end

      subroutine hw2(r1, r2)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      write(*,1000) 'Hello, World! sin(',r1+r2,')=',s
 1000 format(A,F6.3,A,F8.6)
      return
      end

C     special version of hw1 where the result is
C     returned as an argument:

      subroutine hw3_v1(r1, r2, s)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      return
      end

C     F2py treats s as input arg. in hw3_v1; fix this:

      subroutine hw3(r1, r2, s)
      real*8 r1, r2, s
Cf2py intent(out) s
      s = sin(r1 + r2)
      return
      end

C     test case sensitivity:

      subroutine Hw4(R1, R2, s)
      real*8 R1, r2, S
Cf2py intent(out) s
      s = SIN(r1 + r2)
      ReTuRn
      end

C end of F77 file
```

f2py comes automaticaly with numpy; we make a subdirectory f2py-hw and run f2py in this subdirectory:

```
f2py -m hw -c ../hw.f
```

The reuslt is the module hw.so which may be loaded into Python by an ordinary import statement.

- -m option specifies the name of the extension module
- -c indicates that f2py should compile and link the module
- –fcompiler=gfortran specifies the compiler to use (less error messages)
- f2py -c –help-fcompiler to see a list of Fortran compilers installed
- f2py -c –help-compiler to see a list of C compilers installed

Test if everything went fine:

```
python -c 'import hw'
```

We can test now the new module with the previous hwa.py program, who does not know if the module hw is written in Python or Fortran.

When dealing with more complicated libraries, one may want to create Python interfaces to only some of the functions, e.g.:

```
f2py -m hw -c --fcompiler=gfortran ../hw.f only: hw1 hw2 :
```

-h hw.pyf –overwrite-signature options makes f2py to write the module interfaces to a file hw.pyf that you can adjust to your needs:

```
f2py -m hw -h hw.pyf --overwrite-signature ../hw.f
f2py -m hw -h hw.pyf --overwrite-signature ../hw.f only: hw1 hw2 :
```

The function hw3_v1 enables us to see one difficulty of mixing Fortran and Python. The rsult of the computations is stored in the output variable s. Since Fortran employs the call by reference for all the arguments, any change to an argument is visible in the calling code. Let us see how the interface looks like:

Bei dfault, f2py treates r1, r2, and s as input argumnets; calling hw3_v1 shows that the value of the Fortran s variable is not returned to the Python s variable in the call. The remdey is to tell f2py that s is an output parameter. We do this by replacing in the hw.pyf file

```
real*8 :: s
```

with:

```
real*8 intent(out) :: s
```

as in subroutine hw3. The directives intent(in) and intent(out) specify input or output variables, while intent(in,out) are employed for variables for both input and output.

Compiling and linking the hw module with th emodified interface specification in hw.pyf are now performed by:

```
f2py -c --fcompiler=gfortran hw.pyf ../hw.f
```

Note thet f2py changes the interface to become more Pythonic, i.e. we write in Python:

```
s = hw3(r1,r2)
```

For a Fortran routine:

```
subroutine somef(in1, in2, o1, o2, o3, o4, io1)
```

where in1, in2 are input variables, o1, o2, o3, o4 are output variables and io1 is an input/output variable, the generated Python interface will have i1, i2, io1 as input arguments and o1, o2, o3, o4, io1 is returend as a tuple:

```
o1, o2, o3, o4, io1 = somef(i1, i2, io1)
```

As an alternative to the modification of the .pyf file, we may insert an intent specification as a special Cf2py comment in the Fortran source code file. The safest way of writing hw3 is to specify the input/output nature of all the function arguments:

```
      subroutine hw3(r1,r2,s)
      real*8 :: r1, r2, s
Cf2py intent(in) r1
Cf2py intent(in) r2
Cf2py intent(out) s
      s = sin(r1 + r2)
      return
      end
```

Building an Extension Module with Distutils

The standard way for buiilding and installing Python modules uses Python's Distutils tool which comes with the standard Python distribution. An enhanced version of Distutils with better support for Fortran code comes with numpy. We have to create a script setup.py which calls a function steup in Distutils. Then, in order to build the extension module:

```
python setup.py build
```

or to build and install:

```
python steup.py install
```

To build in the current working directory:

```
python setup.py build build_ext --inplace
```

An example of steup.py file

```
1  from numpy.distutils.core import Extension, setup
2
3  setup(name='hw',
4        ext_modules=[Extension(name='hw', sources=['../hw.f'])],
5        )
```