Three ways to wrap - getting started

Wrapping Fortran or C functions to Python using F2PY consists of the following steps:

- Creating the so-called signature file that contains descriptions of wrappers to Fortran or C functions, also called as signatures of the functions. In the case of Fortran routines, F2PY can create initial signature file by scanning Fortran source codes and catching all relevant information needed to create wrapper functions.
- Optionally, F2PY created signature files can be edited to optimize wrappers functions, make them “smarter” and more “Pythonic”.
- F2PY reads a signature file and writes a Python C/API module containing Fortran/C/Python bindings.
- F2PY compiles all sources and builds an extension module containing the wrappers. In building extension modules, F2PY uses `numpy_distutils` that supports a number of Fortran 77/90/95 compilers, including Gnu, Intel, Sun Fortre, SGI MIPSpro, Absoft, NAG, Compaq etc. compilers.

Depending on a particular situation, these steps can be carried out either by just in one command or step-by-step, some steps can be omitted or combined with others.

Below I’ll describe three typical approaches of using F2PY. The following example Fortran 77 code (fib1.f) will be used for illustration:

```
C FILE: FIB1.F
      SUBROUTINE FIB(A,N)
C
C     CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
      DO I=1,N
         IF (I.EQ.1) THEN
            A(I) = 0.0D0
         ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
         ELSE
            A(I) = A(I-1) + A(I-2)
         ENDIF
      ENDDO
      END
C END FILE FIB1.F
```

The quick way

The quickest way to wrap the Fortran subroutine `FIB` to Python is to run

```
f2py -c fib1.f -m fib1
```

This command builds (see `-c` flag, execute `f2py` without arguments to see the explanation of command line options) an extension module `fib1.so` (see `-m` flag) to the current directory. Now, in Python the Fortran subroutine `FIB` is accessible via `fib1.fib`:

```
>>>
>>> import numpy
>>> import fib1
>>> print fib1.fib.__doc__
fib - Function signature:
  fib(a,[n])
Required arguments:
  a : input rank-1 array('d') with bounds (n)
Optional arguments:
  n := len(a) input int

>>> a = numpy.zeros(8,'d')
>>> fib1.fib(a)
>>> print a
[  0.   1.   1.   2.   3.   5.   8.  13.]
```

Note:

- Note that F2PY found that the second argument `n` is the dimension of the first array argument `a`. Since by default all arguments are input-only arguments, F2PY concludes that `n` can be optional with the default value `len(a)`.
- One can use different values for optional `n`:

  ```
  >>>
  >>> a1 = numpy.zeros(8,'d')
  >>> fib1.fib(a1,6)
  >>> print a1
  [ 0.  1.  1.  2.  3.  5.  0.  0.]
  ```

  but an exception is raised when it is incompatible with the input array `a`:

  ```
  >>>
  >>> fib1.fib(a,10)
  fib:n=10
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  fib.error: (len(a)>=n) failed for 1st keyword n
  >>>
  ```

  This demonstrates one of the useful features in F2PY, that it, F2PY implements basic compatibility checks between related arguments in order to avoid any unexpected crashes.
- When a Numpy array, that is Fortran contiguous and has a dtype corresponding to presumed Fortran type, is used as an input array argument, then its C pointer is directly passed to Fortran.

  Otherwise F2PY makes a contiguous copy (with a proper dtype) of the input array and passes C pointer of the copy to Fortran subroutine. As a result, any possible changes to the (copy of) input array have no effect to the original argument, as demonstrated below:

```
>>> a = numpy.ones(8,'i')
>>> fib1.fib(a)
>>> print a
[1 1 1 1 1 1 1 1]
```

Clearly, this is not an expected behaviour. The fact that the above example worked with `dtype=float` is considered accidental.

F2PY provides `intent(inplace)` attribute that would modify the attributes of an input array so that any changes made by Fortran routine will be effective also in input argument. For example, if one specifies `intent(inplace) a` (see below, how), then the example above would read:

```
>>> a = numpy.ones(8,'i')
>>> fib1.fib(a)
>>> print a
[  0.   1.   1.   2.   3.   5.   8.  13.]
```

However, the recommended way to get changes made by Fortran subroutine back to python is to use `intent(out)` attribute. It is more efficient and a cleaner solution.

- The usage of `fib1.fib` in Python is very similar to using `FIB` in Fortran. However, using *in situ* output arguments in Python indicates a poor style as there is no safety mechanism in Python with respect to wrong argument types. When using Fortran or C, compilers naturally discover any type mismatches during compile time but in Python the types must be checked in runtime. So, using *in situ* output arguments in Python may cause difficult to find bugs, not to mention that the codes will be less readable when all required type checks are implemented.

Though the demonstrated way of wrapping Fortran routines to Python is very straightforward, it has several drawbacks (see the comments above). These drawbacks are due to the fact that there is no way that F2PY can determine what is the actual intention of one or the other argument, is it input or output argument, or both, or something else. So, F2PY conservatively assumes that all arguments are input arguments by default.

However, there are ways (see below) how to “teach” F2PY about the true intentions (among other things) of function arguments; and then F2PY is able to generate more Pythonic (more explicit, easier to use, and less error prone) wrappers to Fortran functions.

The smart way

Let’s apply the steps of wrapping Fortran functions to Python one by one.

- First, we create a signature file from `fib1.f` by running

```
f2py fib1.f -m fib2 -h fib1.pyf
```

The signature file is saved to `fib1.pyf` (see `-h` flag) and its contents is shown below.

```
!    -*- f90 -*-
python module fib2 ! in
    interface  ! in :fib2
        subroutine fib(a,n) ! in :fib2:fib1.f
            real*8 dimension(n) :: a
            integer optional,check(len(a)>=n),depend(a) :: n=len(a)
        end subroutine fib
    end interface
end python module fib2

! This file was auto-generated with f2py (version:2.28.198-1366).
! See http://cens.ioc.ee/projects/f2py2e/
```

- Next, we'll teach F2PY that the argument `n` is a input argument (use `intent(in)` attribute) and that the result, i.e. the contents of `a` after calling Fortran function `FIB`, should be returned to Python (use `intent(out)` attribute). In addition, an array `a` should be created dynamically using the size given by the input argument `n` (use `depend(n)` attribute to indicate dependence relation).

  The content of a modified version of `fib1.pyf` (saved as `fib2.pyf`) is as follows:

```
!    -*- f90 -*-
python module fib2
    interface
        subroutine fib(a,n)
            real*8 dimension(n),intent(out),depend(n) :: a
            integer intent(in) :: n
        end subroutine fib
    end interface
end python module fib2
```

- And finally, we build the extension module by running

```
f2py -c fib2.pyf fib1.f
```

In Python:

```
                                                                    >>>
>>> import fib2
>>> print fib2.fib.__doc__
fib - Function signature:
  a = fib(n)
Required arguments:
  n : input int
Return objects:
  a : rank-1 array('d') with bounds (n)

>>> print fib2.fib(8)
[  0.   1.   1.   2.   3.   5.   8.  13.]
```

Note:

- Clearly, the signature of `fib2.fib` now corresponds to the intention of Fortran subroutine `FIB` more closely: given the number `n`, `fib2.fib` returns the first `n` Fibonacci numbers as a

Numpy array. Also, the new Python signature `fib2.fib` rules out any surprises that we experienced with `fib1.fib`.
- Note that by default using single `intent(out)` also implies `intent(hide)`. Argument that has `intent(hide)` attribute specified, will not be listed in the argument list of a wrapper function.

The quick and smart way

The “smart way” of wrapping Fortran functions, as explained above, is suitable for wrapping (e.g. third party) Fortran codes for which modifications to their source codes are not desirable nor even possible.

However, if editing Fortran codes is acceptable, then the generation of an intermediate signature file can be skipped in most cases. Namely, F2PY specific attributes can be inserted directly to Fortran source codes using the so-called F2PY directive. A F2PY directive defines special comment lines (starting with `Cf2py`, for example) which are ignored by Fortran compilers but F2PY interprets them as normal lines.

Here is shown a modified version of the example Fortran code (fib3.f), saved as `fib3.f`:

```
C FILE: FIB3.F
      SUBROUTINE FIB(A,N)
C
C     CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
Cf2py intent(in) n
Cf2py intent(out) a
Cf2py depend(n) a
      DO I=1,N
         IF (I.EQ.1) THEN
            A(I) = 0.0D0
         ELSEIF (I.EQ.2) THEN
            A(I) = 1.0D0
         ELSE
            A(I) = A(I-1) + A(I-2)
         ENDIF
      ENDDO
      END
C END FILE FIB3.F
```

Building the extension module can be now carried out in one command:

```
f2py -c -m fib3 fib3.f
```

Notice that the resulting wrapper to `FIB` is as “smart” as in previous case:

```
>>> import fib3
>>> print fib3.fib.__doc__
fib - Function signature:
  a = fib(n)
Required arguments:
  n : input int
Return objects:
  a : rank-1 array('d') with bounds (n)

>>> print fib3.fib(8)
[  0.   1.   1.   2.   3.   5.   8.  13.]
```

Table Of Contents (../contents.html)

Previous topic

F2PY Users Guide and Reference Manual (index.html)

Next topic

Signature file (signature-file.html)