F2PY Users Guide and Reference Manual

Author: Pearu Peterson
Contact: pearu@cens.ioc.ee
Web site: http://cens.ioc.ee/projects/f2py2e/
Date: 2005-01-30
Revision: 1.25

Contents

1 Introduction

The purpose of the F2PY –*Fortran to Python interface generator*– project is to provide a connection between Python and Fortran languages. F2PY is a Python package (with a command line tool `f2py` and a module `f2py2e`) that facilitates creating/building Python C/API extension modules that make it possible

- to call Fortran 77/90/95 external subroutines and Fortran 90/95 module subroutines as well as C functions;
- to access Fortran 77 `COMMON` blocks and Fortran 90/95 module data, including allocatable arrays

from Python. See F2PY web site for more information and installation instructions.

2 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 `scipy_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 ommited or combined with others.

Below I'll describe three typical approaches of using F2PY. The following example Fortran 77 code 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
```

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

Comments

- 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=Numeric.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 Numeric array, that is Fortran contiguous and has a typecode 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 typecode) 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=Numeric.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 `typecode='d'` 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=Numeric.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 acctual 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.

2.2 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.]
```

Comments

- 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 Numeric 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.

2.3 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, 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.]
```

3 Signature file

The syntax specification for signature files (.pyf files) is borrowed from the Fortran 90/95 language specification. Almost all Fortran 90/95 standard constructs are understood, both in free and fixed format (recall that Fortran 77 is a subset of Fortran 90/95). F2PY introduces also some extensions to Fortran 90/95 language specification that help designing Fortran to Python interface, make it more "Pythonic".

Signature files may contain arbitrary Fortran code (so that Fortran codes can be considered as signature files). F2PY silently ignores Fortran constructs that are irrelevant for creating the interface. However, this includes also syntax errors. So, be careful not making ones;-).

In general, the contents of signature files is case-sensitive. When scanning Fortran codes and writing a signature file, F2PY lowers all cases automatically except in multi-line blocks or when `--no-lower` option is used.

The syntax of signature files is overvied below.

3.1 Python module block

A signature file may contain one (recommended) or more `python module` blocks. `python module` block describes the contents of a Python/C extension module `<modulename>module.c` that F2PY generates.

Exception: if `<modulename>` contains a substring `__user__`, then the corresponding `python module` block describes the signatures of so-called call-back functions (see Call-back arguments).

A `python module` block has the following structure:

```
python module <modulename>
  [<usercode statement>]...
  [
  interface
    <usercode statement>
    <Fortran block data signatures>
    <Fortran/C routine signatures>
  end [interface]
  ]...
  [
  interface
    module <F90 modulename>
      [<F90 module data type declarations>]
      [<F90 module routine signatures>]
    end [module [<F90 modulename>]]
  end [interface]
  ]...
end [python module [<modulename>]]
```

Here brackets `[]` indicate a optional part, dots `...` indicate one or more of a previous part. So, `[]...` reads zero or more of a previous part.

3.2 Fortran/C routine signatures

The signature of a Fortran routine has the following structure:

```
[<typespec>] function | subroutine <routine name> \
              [ ( [<arguments>] ) ] [ result ( <entityname> ) ]
  [<argument/variable type declarations>]
  [<argument/variable attribute statements>]
  [<use statements>]
  [<common block statements>]
  [<other statements>]
end [ function | subroutine [<routine name>] ]
```

From a Fortran routine signature F2PY generates a Python/C extension function that has the following signature:

```
def <routine name>(<required arguments>[,<optional arguments>]):
    ...
    return <return variables>
```

The signature of a Fortran block data has the following structure:

```
block data [ <block data name> ]
  [<variable type declarations>]
  [<variable attribute statements>]
  [<use statements>]
  [<common block statements>]
  [<include statements>]
end [ block data [<block data name>] ]
```

3.2.1 Type declarations

The definition of the `<argument/variable type declaration>` part is

```
<typespec> [ [<attrspec>] :: ] <entitydecl>
```

where

```
<typespec> := byte | character [<charselector>]
           | complex [<kindselector>] | real [<kindselector>]
           | double complex | double precision
           | integer [<kindselector>] | logical [<kindselector>]

<charselector> := * <charlen>
               | ( [len=] <len> [ , [kind=] <kind>] )
               | ( kind= <kind> [ , len= <len> ] )
<kindselector> := * <intlen> | ( [kind=] <kind> )

<entitydecl> := <name> [ [ * <charlen> ] [ ( <arrayspec> ) ]
                     | [ ( <arrayspec> ) ] * <charlen> ]
                    | [ / <init_expr> / | = <init_expr> ] \
                      [ , <entitydecl> ]
```

and

- `<attrspec>` is a comma separated list of attributes;
- `<arrayspec>` is a comma separated list of dimension bounds;
- `<init_expr>` is a C expression.
- `<intlen>` may be negative integer for `integer` type specifications. In such cases `integer*<negintlen>` represents unsigned C integers.

If an argument has no `<argument type declaration>`, its type is determined by applying `implicit` rules to its name.

3.2.2 Statements

Attribute statements:

The `<argument/variable attribute statement>` is `<argument/variable type declaration>` without `<typespec>`. In addition, in an attribute statement one cannot use other attributes, also `<entitydecl>` can be only a list of names.

Use statements:

The definition of the `<use statement>` part is

```
use <modulename> [ , <rename_list> | , ONLY : <only_list> ]
```

where

```
<rename_list> := <local_name> => <use_name> [ , <rename_list> ]
```

Currently F2PY uses `use` statement only for linking call-back modules and `external` arguments (call-back functions), see Call-back arguments.

Common block statements:

The definition of the `<common block statement>` part is

```
common / <common name> / <shortentitydecl>
```

where

```
<shortentitydecl> := <name> [ ( <arrayspec> ) ] [ , <shortentitydecl> ]
```

One `python module` block should not contain two or more `common` blocks with the same name. Otherwise, the latter ones are ignored. The types of variables in `<shortentitydecl>` are defined using `<argument type declarations>`. Note that the corresponding `<argument type declarations>` may contain array specifications; then you don't need to specify these in `<shortentitydecl>`.

Other statements:

The `<other statement>` part refers to any other Fortran language constructs that are not described above. F2PY ignores most of them except

- `call` statements and function calls of `external` arguments (more details?);
- `include` statements

  ```
  include '<filename>'
  include "<filename>"
  ```

  If a file `<filename>` does not exist, the `include` statement is ignored. Otherwise, the file `<filename>` is included to a signature file. `include` statements can be used in any part of a signature file, also outside the Fortran/C routine signature blocks.
- `implicit` statements

  ```
  implicit none
  implicit <list of implicit maps>
  ```

  where

  ```
  <implicit map> := <typespec> ( <list of letters or range of letters> )
  ```

  Implicit rules are used to deterimine the type specification of a variable (from the first-letter of its name) if the variable is not defined using `<variable type declaration>`. Default implicit rule is given by

  ```
  implicit real (a-h,o-z,$_), integer (i-m)
  ```
- `entry` statements

  ```
  entry <entry name> [([<arguments>])]
  ```

  F2PY generates wrappers to all entry names using the signature of the routine block. Tip: `entry` statement can be used to describe the signature of an arbitrary routine allowing F2PY to generate a number of wrappers from only one routine block signature. There are few restrictions while doing this: `fortranname` cannot be used, `callstatement` and `callprotoargument` can be used only if they are valid for all entry routines, etc.

In addition, F2PY introduces the following statements:

- `threadsafe` Use `Py_BEGIN_ALLOW_THREADS .. Py_END_ALLOW_THREADS` block around the call to Fortran/C function.
- `callstatement <C-expr|multi-line block>` Replace F2PY generated call statement to Fortran/C function with `<C-expr|multi-line block>`. The wrapped Fortran/C function is available as `(*f2py_func)`. To raise an exception, set `f2py_success = 0` in `<C-expr|multi-line block>`.
- `callprotoargument <C-typespecs>` When `callstatement` statement is used then F2PY may not generate proper prototypes for Fortran/C functions (because `<C-expr>` may contain any function calls and F2PY has no way to determine what should be the proper prototype). With this statement you can explicitely specify the arguments of the corresponding prototype:

  ```
  extern <return type> FUNC_F(<routine name>,<ROUTINE NAME>)(<callprotoargument>);
  ```

- `fortranname [<acctual Fortran/C routine name>]` You can use arbitrary `<routine name>` for a given Fortran/C function. Then you have to specify `<acctual Fortran/C routine name>` with this statement.
  If `fortranname` statement is used without `<acctual Fortran/C routine name>` then a dummy wrapper is generated.
- `usercode <multi-line block>` When used inside `python module` block, then given C code will be inserted to generated C/API source just before wrapper function definitions. Here you can define arbitrary C functions to be used in initialization of optional arguments, for example. If `usercode` is used twise inside `python module` block then the second multi-line block is inserted after the definition of external routines.
  When used inside `<routine singature>`, then given C code will be inserted to the corresponding wrapper function just after declaring variables but before any C statements. So, `usercode` follow-up can contain both declarations and C statements.
  When used inside the first `interface` block, then given C code will be inserted at the end of the initialization function of the extension module. Here you can modify extension modules dictionary. For example, for defining additional variables etc.
- `pymethoddef <multi-line block>` Multiline block will be inserted to the definition of module methods `PyMethodDef`-array. It must be a comma-separated list of C arrays (see Extending and Embedding Python documentation for details). `pymethoddef` statement can be used only inside `python module` block.

3.2.3 Attributes

The following attributes are used by F2PY:

`optional` The corresponding argument is moved to the end of `<optional arguments>` list. A default value for an optional argument can be specified `<init_expr>`, see `entitydecl` definition. Note that the default value must be given as a valid C expression.

Note that whenever `<init_expr>` is used, `optional` attribute is set automatically by F2PY.

For an optional array argument, all its dimensions must be bounded.

`required` The corresponding argument is considered as a required one. This is default. You need to specify `required` only if there is a need to disable automatic `optional` setting when `<init_expr>` is used.

If Python `None` object is used as an required argument, the argument is treated as optional. That is, in the case of array argument, the memory is allocated. And if `<init_expr>` is given, the corresponding initialization is carried out.

`dimension(<arrayspec>)` The corresponding variable is considered as an array with given dimensions in `<arrayspec>`.

`intent(<intentspec>)` This specifies the "intention" of the corresponding argument. `<intentspec>` is a comma separated list of the following keys:

- `in` The argument is considered as an input-only argument. It means that the value of the argument is passed to Fortran/C function and that function is expected not to change the value of an argument.

- `inout` The argument is considered as an input/output or *in situ* output argument. `intent(inout)` arguments can be only "contiguous" Numeric arrays with proper type and size. Here "contiguous" can be either in Fortran or C sense. The latter one coincides with the contiguous concept used in Numeric and is effective only if `intent(c)` is used. Fortran-contiguousness is assumed by default.
  Using `intent(inout)` is generally not recommended, use `intent(in,out)` instead. See also `intent(inplace)` attribute.
- `inplace` The argument is considered as an input/output or *in situ* output argument. `intent(inplace)` arguments must be Numeric arrays with proper size. If the type of an array is not "proper" or the array is non-contiguous then the array will be changed in-place to fix the type and make it contiguous.
  Using `intent(inplace)` is generally not recommended either. For example, when slices have been taken from an `intent(inplace)` argument then after in-place changes, slices data pointers may point to unallocated memory area.
- `out` The argument is considered as an return variable. It is appended to the `<returned variables>` list. Using `intent(out)` sets `intent(hide)` automatically, unless also `intent(in)` or `intent(inout)` were used.
  By default, returned multidimensional arrays are Fortran-contiguous. If `intent(c)` is used, then returned multi-dimensional arrays are C-contiguous.
- `hide` The argument is removed from the list of required or optional arguments. Typically `intent(hide)` is used with `intent(out)` or when `<init_expr>` completely determines the value of the argument like in the following example:

  ```
  integer intent(hide),depend(a) :: n = len(a)
  real intent(in),dimension(n) :: a
  ```
- `c` The argument is treated as a C scalar or C array argument. In the case of a scalar argument, its value is passed to C function as a C scalar argument (recall that Fortran scalar arguments are actually C pointer arguments). In the case of an array argument, the wrapper function is assumed to treat multi-dimensional arrays as C-contiguous arrays.
  There is no need to use `intent(c)` for one-dimensional arrays, no matter if the wrapped function is either a Fortran or a C function. This is because the concepts of Fortran- and C-contiguousness overlap in one-dimensional cases.
  If `intent(c)` is used as an statement but without entity declaration list, then F2PY adds `intent(c)` attibute to all arguments.
  Also, when wrapping C functions, one must use `intent(c)` attribute for `<routine name>` in order to disable Fortran specific `F_FUNC(..,..)` macros.
- `cache` The argument is treated as a junk of memory. No Fortran nor C contiguousness checks are carried out. Using `intent(cache)` makes sense only for array arguments, also in connection with `intent(hide)` or `optional` attributes.
- `copy` Ensure that the original contents of `intent(in)` argument is preserved. Typically used in connection with `intent(in,out)` attribute. F2PY creates an optional argument `overwrite_<argument name>` with the default value `0`.
- `overwrite` The original contents of the `intent(in)` argument may be altered by the Fortran/C function. F2PY creates an optional argument `overwrite_<argument name>` with the default value `1`.
- `out=<new name>` Replace the return name with `<new name>` in the `__doc__` string of a wrapper function.
- `callback` Construct an external function suitable for calling Python function from Fortran. `intent(callback)` must be specified before the corresponding `external` statement. If 'argument' is not in argument list then it will be added to Python wrapper but is not used when calling Fortran function.
  Use `intent(callback)` in situations where a Fortran/C code assumes that a user implements a function with given prototype and links it to an executable.
- `aux` Define auxiliary C variable in F2PY generated wrapper function. Useful to save parameter values so that they can be accessed in initialization expression of other variables. Note that `intent(aux)` silently implies `intent(c)`.

The following rules apply:

- If no `intent(in | inout | out | hide)` is specified, `intent(in)` is assumed.

- `intent(in,inout)` is `intent(in)`.
- `intent(in,hide)` or `intent(inout,hide)` is `intent(hide)`.
- `intent(out)` is `intent(out,hide)` unless `intent(in)` or `intent(inout)` is specified.
- If `intent(copy)` or `intent(overwrite)` is used, then an additional optional argument is introduced with a name `overwrite_<argument name>` and a default value 0 or 1, respectively.
- `intent(inout,inplace)` is `intent(inplace)`.
- `intent(in,inplace)` is `intent(inplace)`.

`check([<C-booleanexpr>])` Perform consistency check of arguments by evaluating `<C-booleanexpr>`; if `<C-booleanexpr>` returns 0, an exception is raised.

If `check(..)` is not used then F2PY generates few standard checks (e.g. in a case of an array argument, check for the proper shape and size) automatically. Use `check()` to disable checks generated by F2PY.

`depend([<names>])` This declares that the corresponding argument depends on the values of variables in the list `<names>`. For example, `<init_expr>` may use the values of other arguments. Using information given by `depend(..)` attributes, F2PY ensures that arguments are initialized in a proper order. If `depend(..)` attribute is not used then F2PY determines dependence relations automatically. Use `depend()` to disable dependence relations generated by F2PY.

When you edit dependence relations that were initially generated by F2PY, be careful not to break the dependence relations of other relevant variables. Another thing to watch out is cyclic dependencies. F2PY is able to detect cyclic dependencies when constructing wrappers and it complains if any are found.

`allocatable` The corresponding variable is Fortran 90 allocatable array defined as Fortran 90 module data.

`external` The corresponding argument is a function provided by user. The signature of this so-called call-back function can be defined

- in `__user__` module block,
- or by demonstrative (or real, if the signature file is a real Fortran code) call in the `<other statements>` block.

For example, F2PY generates from

```
external cb_sub, cb_fun
integer n
real a(n),r
call cb_sub(a,n)
r = cb_fun(4)
```

the following call-back signatures:

```
subroutine cb_sub(a,n)
    real dimension(n) :: a
    integer optional,check(len(a)>=n),depend(a) :: n=len(a)
end subroutine cb_sub
function cb_fun(e_4_e) result (r)
    integer :: e_4_e
    real :: r
end function cb_fun
```

The corresponding user-provided Python function are then:

```
def cb_sub(a,[n]):
    ...
    return
def cb_fun(e_4_e):
    ...
    return r
```

See also `intent(callback)` attribute.

`parameter` The corresponding variable is a parameter and it must have a fixed value. F2PY replaces all parameter occurrences by their corresponding values.

3.3 Extensions

3.3.1 F2PY directives

The so-called F2PY directives allow using F2PY signature file constructs also in Fortran 77/90 source codes. With this feature you can skip (almost) completely intermediate signature file generations and apply F2PY directly to Fortran source codes.

F2PY directive has the following form:

```
<comment char>f2py ...
```

where allowed comment characters for fixed and free format Fortran codes are `cC*!#` and `!`, respectively. Everything that follows `<comment char>f2py` is ignored by a compiler but read by F2PY as a normal Fortran (non-comment) line:

> When F2PY finds a line with F2PY directive, the directive is first replaced by 5 spaces and then the line is reread.

For fixed format Fortran codes, `<comment char>` must be at the first column of a file, of course. For free format Fortran codes, F2PY directives can appear anywhere in a file.

3.3.2 C expressions

C expressions are used in the following parts of signature files:

- `<init_expr>` of variable initialization;
- `<C-booleanexpr>` of the `check` attribute;
- `<arrayspec> of the ‘‘dimension` attribute;
- `callstatement` statement, here also a C multi-line block can be used.

A C expression may contain:

- standard C constructs;
- functions from `math.h` and `Python.h`;
- variables from the argument list, presumably initialized before according to given dependence relations;
- the following CPP macros:

  `rank(<name>)` Returns the rank of an array `<name>`.

  `shape(<name>,<n>)` Returns the `<n>`-th dimension of an array `<name>`.

  `len(<name>)` Returns the lenght of an array `<name>`.

  `size(<name>)` Returns the size of an array `<name>`.

  `slen(<name>)` Returns the length of a string `<name>`.

For initializing an array `<array name>`, F2PY generates a loop over all indices and dimensions that executes the following pseudo-statement:

```
<array name>(_i[0],_i[1],...) = <init_expr>;
```

where `_i[<i>]` refers to the `<i>`-th index value and that runs from `0` to `shape(<array name>,<i>)-1`.

For example, a function `myrange(n)` generated from the following signature

```
subroutine myrange(a,n)
  fortranname        ! myrange is a dummy wrapper
  integer intent(in) :: n
  real*8 intent(c,out),dimension(n),depend(n) :: a = _i[0]
end subroutine myrange
```

is equivalent to `Numeric.arange(n,typecode='d')`.

Warning!

F2PY may lower cases also in C expressions when scanning Fortran codes (see `--[no]-lower` option).

3.3.3 Multi-line blocks

A multi-line block starts with ''' (triple single-quotes) and ends with ''' in some *strictly* subsequent line. Multi-line blocks can be used only within .pyf files. The contents of a multi-line block can be arbitrary (except that it cannot contain ''') and no transformations (e.g. lowering cases) are applied to it.

Currently, multi-line blocks can be used in the following constructs:

- as a C expression of the `callstatement` statement;
- as a C type specification of the `callprotoargument` statement;
- as a C code block of the `usercode` statement;
- as a list of C arrays of the `pymethoddef` statement;
- as documentation string.

4 Using F2PY bindings in Python

All wrappers (to Fortran/C routines or to common blocks or to Fortran 90 module data) generated by F2PY are exposed to Python as `fortran` type objects. Routine wrappers are callable `fortran` type objects while wrappers to Fortran data have attributes referring to data objects.

All `fortran` type object have attribute `_cpointer` that contains CObject referring to the C pointer of the corresponding Fortran/C function or variable in C level. Such CObjects can be used as an callback argument of F2PY generated functions to bypass Python C/API layer of calling Python functions from Fortran or C when the computational part of such functions is implemented in C or Fortran and wrapped with F2PY (or any other tool capable of providing CObject of a function).

Example

Consider a Fortran 77 file `ftype.f`:

```
C FILE: FTYPE.F
      SUBROUTINE FOO(N)
      INTEGER N
Cf2py integer optional,intent(in) :: n = 13
      REAL A,X
      COMMON /DATA/ A,X(3)
      PRINT*, "IN FOO: N=",N," A=",A," X=[",X(1),X(2),X(3),"]"
      END
C END OF FTYPE.F
```

and build a wrapper using:

```
f2py -c ftype.f -m ftype
```

In Python:

```
>>> import ftype
>>> print ftype.__doc__
This module 'ftype' is auto-generated with f2py (version:2.28.198-1366).
Functions:
  foo(n=13)
COMMON blocks:
  /data/ a,x(3)
.
>>> type(ftype.foo),type(ftype.data)
(<type 'fortran'>, <type 'fortran'>)
>>> ftype.foo()
 IN FOO: N= 13 A=  0. X=[  0.  0.  0.]
>>> ftype.data.a = 3
>>> ftype.data.x = [1,2,3]
>>> ftype.foo()
```

```
 IN FOO: N= 13 A=  3. X=[  1.  2.  3.]
>>> ftype.data.x[1] = 45
>>> ftype.foo(24)
 IN FOO: N= 24 A=  3. X=[  1.  45.  3.]
>>> ftype.data.x
array([  1.,  45.,   3.],'f')
```

4.1 Scalar arguments

In general, a scalar argument of a F2PY generated wrapper function can be ordinary Python scalar (integer, float, complex number) as well as an arbitrary sequence object (list, tuple, array, string) of scalars. In the latter case, the first element of the sequence object is passed to Fortran routine as a scalar argument.

Note that when type-casting is required and there is possible loss of information (e.g. when type-casting float to integer or complex to float), F2PY does not raise any exception. In complex to real type-casting only the real part of a complex number is used.

`intent(inout)` scalar arguments are assumed to be array objects in order to *in situ* changes to be effective. It is recommended to use arrays with proper type but also other types work.

Example

Consider the following Fortran 77 code:

```
C FILE: SCALAR.F
      SUBROUTINE FOO(A,B)
      REAL*8 A, B
Cf2py intent(in) a
Cf2py intent(inout) b
      PRINT*, "    A=",A," B=",B
      PRINT*, "INCREMENT A AND B"
      A = A + 1D0
      B = B + 1D0
      PRINT*, "NEW A=",A," B=",B
      END
C END OF FILE SCALAR.F
```

and wrap it using `f2py -c -m scalar scalar.f`.
In Python:

```
>>> import scalar
>>> print scalar.foo.__doc__
foo - Function signature:
  foo(a,b)
Required arguments:
  a : input float
  b : in/output rank-0 array(float,'d')

>>> scalar.foo(2,3)
     A=  2. B=  3.
 INCREMENT A AND B
 NEW A=  3. B=  4.
>>> import Numeric
>>> a=Numeric.array(2)   # these are integer rank-0 arrays
>>> b=Numeric.array(3)
>>> scalar.foo(a,b)
     A=  2. B=  3.
 INCREMENT A AND B
 NEW A=  3. B=  4.
>>> print a,b            # note that only b is changed in situ
2 4
```

4.2 String arguments

F2PY generated wrapper functions accept (almost) any Python object as a string argument, `str` is applied for non-string objects. Exceptions are Numeric arrays that must have type code `'c'` or `'1'` when used as string arguments.

A string can have arbitrary length when using it as a string argument to F2PY generated wrapper function. If the length is greater than expected, the string is truncated. If the length is smaller that expected, additional memory is allocated and filled with `\0`.

Because Python strings are immutable, an `intent(inout)` argument expects an array version of a string in order to *in situ* changes to be effective.

Example

Consider the following Fortran 77 code:

```
C FILE: STRING.F
      SUBROUTINE FOO(A,B,C,D)
      CHARACTER*5 A, B
      CHARACTER*(*) C,D
Cf2py intent(in) a,c
Cf2py intent(inout) b,d
      PRINT*, "A=",A
      PRINT*, "B=",B
      PRINT*, "C=",C
      PRINT*, "D=",D
      PRINT*, "CHANGE A,B,C,D"
      A(1:1) = 'A'
      B(1:1) = 'B'
      C(1:1) = 'C'
      D(1:1) = 'D'
      PRINT*, "A=",A
      PRINT*, "B=",B
      PRINT*, "C=",C
      PRINT*, "D=",D
      END
C END OF FILE STRING.F
```

and wrap it using `f2py -c -m mystring string.f`.

Python session:

```
>>> import mystring
>>> print mystring.foo.__doc__
foo - Function signature:
  foo(a,b,c,d)
Required arguments:
  a : input string(len=5)
  b : in/output rank-0 array(string(len=5),'c')
  c : input string(len=-1)
  d : in/output rank-0 array(string(len=-1),'c')

>>> import Numeric
>>> a=Numeric.array('123')
>>> b=Numeric.array('123')
>>> c=Numeric.array('123')
>>> d=Numeric.array('123')
>>> mystring.foo(a,b,c,d)
 A=123
 B=123
 C=123
 D=123
 CHANGE A,B,C,D
 A=A23
```

```
 B=B23
 C=C23
 D=D23
>>> a.tostring(),b.tostring(),c.tostring(),d.tostring()
('123', 'B23', '123', 'D23')
```

4.3 Array arguments

In general, array arguments of F2PY generated wrapper functions accept arbitrary sequences that can be transformed to Numeric array objects. An exception is `intent(inout)` array arguments that always must be proper-contiguous and have proper type, otherwise an exception is raised. Another exception is `intent(inplace)` array arguments that attributes will be changed in-situ if the argument has different type than expected (see `intent(inplace)` attribute for more information).

In general, if a Numeric array is proper-contiguous and has a proper type then it is directly passed to wrapped Fortran/C function. Otherwise, an element-wise copy of an input array is made and the copy, being proper-contiguous and with proper type, is used as an array argument.

There are two types of proper-contiguous Numeric arrays:

- Fortran-contiguous arrays when data is stored column-wise, i.e. indexing of data as stored in memory starts from the lowest dimension;
- C-contiguous or simply contiguous arrays when data is stored row-wise, i.e. indexing of data as stored in memory starts from the highest dimension.

For one-dimensional arrays these notions coincide.

For example, an 2x2 array `A` is Fortran-contiguous if its elements are stored in memory in the following order:

```
A[0,0] A[1,0] A[0,1] A[1,1]
```

and C-contiguous if the order is as follows:

```
A[0,0] A[0,1] A[1,0] A[1,1]
```

To test whether an array is C-contiguous, use `.iscontiguous()` method of Numeric arrays. To test for Fortran-contiguousness, all F2PY generated extension modules provide a function `has_column_major_storage(<array>)`. This function is equivalent to `Numeric.transpose(<array>).iscontiguous()` but more efficient.

Usually there is no need to worry about how the arrays are stored in memory and whether the wrapped functions, being either Fortran or C functions, assume one or another storage order. F2PY automatically ensures that wrapped functions get arguments with proper storage order; the corresponding algorithm is designed to make copies of arrays only when absolutely necessary. However, when dealing with very large multi-dimensional input arrays with sizes close to the size of the physical memory in your computer, then a care must be taken to use always proper-contiguous and proper type arguments.

To transform input arrays to column major storage order before passing them to Fortran routines, use a function `as_column_major_storage(<array>)` that is provided by all F2PY generated extension modules.

Example

Consider Fortran 77 code:

```
C FILE: ARRAY.F
      SUBROUTINE FOO(A,N,M)
C
C     INCREMENT THE FIRST ROW AND DECREMENT THE FIRST COLUMN OF A
C
      INTEGER N,M,I,J
      REAL*8 A(N,M)
Cf2py intent(in,out,copy) a
Cf2py integer intent(hide),depend(a) :: n=shape(a,0), m=shape(a,1)
      DO J=1,M
         A(1,J) = A(1,J) + 1D0
      ENDDO
      DO I=1,N
```

```
         A(I,1) = A(I,1) - 1D0
       ENDDO
       END
C END OF FILE ARRAY.F
```

and wrap it using `f2py -c -m arr array.f -DF2PY_REPORT_ON_ARRAY_COPY=1`.
In Python:

```
>>> import arr
>>> from Numeric import array
>>> print arr.foo.__doc__
foo - Function signature:
  a = foo(a,[overwrite_a])
Required arguments:
  a : input rank-2 array('d') with bounds (n,m)
Optional arguments:
  overwrite_a := 0 input int
Return objects:
  a : rank-2 array('d') with bounds (n,m)

>>> a=arr.foo([[1,2,3],
...            [4,5,6]])
copied an array using PyArray_CopyFromObject: size=6, elsize=8
>>> print a
[[ 1.  3.  4.]
 [ 3.  5.  6.]]
>>> a.iscontiguous(), arr.has_column_major_storage(a)
(0, 1)
>>> b=arr.foo(a)              # even if a is proper-contiguous
...                           # and has proper type, a copy is made
...                           # forced by intent(copy) attribute
...                           # to preserve its original contents
...
copied an array using copy_ND_array: size=6, elsize=8
>>> print a
[[ 1.  3.  4.]
 [ 3.  5.  6.]]
>>> print b
[[ 1.  4.  5.]
 [ 2.  5.  6.]]
>>> b=arr.foo(a,overwrite_a=1) # a is passed directly to Fortran
...                            # routine and its contents is discarded
...
>>> print a
[[ 1.  4.  5.]
 [ 2.  5.  6.]]
>>> print b
[[ 1.  4.  5.]
 [ 2.  5.  6.]]
>>> a is b                      # a and b are acctually the same objects
1
>>> print arr.foo([1,2,3])      # different rank arrays are allowed
copied an array using PyArray_CopyFromObject: size=3, elsize=8
[ 1.  1.  2.]
>>> print arr.foo([[[1],[2],[3]]])
copied an array using PyArray_CopyFromObject: size=3, elsize=8
[ [[ 1.]
  [ 3.]
  [ 4.]]]
>>>
>>> # Creating arrays with column major data storage order:
```

```
...
>>> s = arr.as_column_major_storage(array([[1,2,3],[4,5,6]]))
copied an array using copy_ND_array: size=6, elsize=4
>>> arr.has_column_major_storage(s)
1
>>> print s
[[1 2 3]
 [4 5 6]]
>>> s2 = arr.as_column_major_storage(s)
>>> s2 is s    # an array with column major storage order
               # is returned immediately
1
```

4.4 Call-back arguments

F2PY supports calling Python functions from Fortran or C codes.

Example

Consider the following Fortran 77 code

```
C FILE: CALLBACK.F
      SUBROUTINE FOO(FUN,R)
      EXTERNAL FUN
      INTEGER I
      REAL*8 R
Cf2py intent(out) r
      R = 0D0
      DO I=-5,5
         R = R + FUN(I)
      ENDDO
      END
C END OF FILE CALLBACK.F
```

and wrap it using `f2py -c -m callback callback.f`.
In Python:

```
>>> import callback
>>> print callback.foo.__doc__
foo - Function signature:
  r = foo(fun,[fun_extra_args])
Required arguments:
  fun : call-back function
Optional arguments:
  fun_extra_args := () input tuple
Return objects:
  r : float
Call-back functions:
  def fun(i): return r
  Required arguments:
    i : input int
  Return objects:
    r : float

>>> def f(i): return i*i
...
>>> print callback.foo(f)
110.0
>>> print callback.foo(lambda i:1)
11.0
```

In the above example F2PY was able to guess accurately the signature of a call-back function. However, sometimes F2PY cannot establish the signature as one would wish and then the signature of a call-back function must be modified in the signature file manually. Namely, signature files may contain special modules (the names of such modules contain a substring `__user__`) that collect various signatures of call-back functions. Callback arguments in routine signatures have attribute `external` (see also `intent(callback)` attribute). To relate a callback argument and its signature in `__user__` module block, use `use` statement as illustrated below. The same signature of a callback argument can be referred in different routine signatures.

Example

We use the same Fortran 77 code as in previous example but now we'll pretend that F2PY was not able to guess the signatures of call-back arguments correctly. First, we create an initial signature file `callback2.pyf` using F2PY:

```
f2py -m callback2 -h callback2.pyf callback.f
```

Then modify it as follows

```
!    -*- f90 -*-
python module __user__routines
    interface
        function fun(i) result (r)
            integer :: i
            real*8 :: r
        end function fun
    end interface
end python module __user__routines

python module callback2
    interface
        subroutine foo(f,r)
            use __user__routines, f=>fun
            external f
            real*8 intent(out) :: r
        end subroutine foo
    end interface
end python module callback2
```

Finally, build the extension module using:

```
f2py -c callback2.pyf callback.f
```

An example Python session would be identical to the previous example except that argument names would differ.

Sometimes a Fortran package may require that users provide routines that the package will use. F2PY can construct an interface to such routines so that Python functions could be called from Fortran.

Example

Consider the following Fortran 77 subroutine that takes an array and applies a function `func` to its elements.

```
      subroutine calculate(x,n)
cf2py intent(callback) func
      external func
c     The following lines define the signature of func for F2PY:
cf2py real*8 y
cf2py y = func(y)
c
cf2py intent(in,out,copy) x
      integer n,i
      real*8 x(n)
      do i=1,n
         x(i) = func(x(i))
```

```
      end do
      end
```

It is expected that function `func` has been defined externally. In order to use a Python function as `func`, it must have an attribute `intent(callback)` (it must be specified before the `external` statement).

Finally, build an extension module using:

```
f2py -c -m foo calculate.f
```

In Python:

```
>>> import foo
>>> foo.calculate(range(5), lambda x: x*x)
array([  0.,   1.,   4.,   9.,  16.])
>>> import math
>>> foo.calculate(range(5), math.exp)
array([  1.        ,   2.71828175,   7.38905621,  20.08553696,  54.59814835])
```

F2PY generated interface is very flexible with respect to call-back arguments. For each call-back argument an additional optional argument `<name>_extra_args` is introduced by F2PY. This argument can be used to pass extra arguments to user provided call-back arguments.

If a F2PY generated wrapper function expects the following call-back argument:

```
def fun(a_1,...,a_n):
   ...
   return x_1,...,x_k
```

but the following Python function

```
def gun(b_1,...,b_m):
   ...
   return y_1,...,y_l
```

is provided by an user, and in addition,

```
fun_extra_args = (e_1,...,e_p)
```

is used, then the following rules are applied when a Fortran or C function calls the call-back argument `gun`:

- If `p==0` then `gun(a_1,...,a_q)` is called, here `q=min(m,n)`.
- If `n+p<=m` then `gun(a_1,...,a_n,e_1,...,e_p)` is called.
- If `p<=m<n+p` then `gun(a_1,...,a_q,e_1,...,e_p)` is called, here `q=m-p`.
- If `p>m` then `gun(e_1,...,e_m)` is called.
- If `n+p` is less than the number of required arguments to `gun` then an exception is raised.

The function `gun` may return any number of objects as a tuple. Then following rules are applied:

- If `k<l`, then `y_{k+1},...,y_l` are ignored.
- If `k>l`, then only `x_1,...,x_l` are set.

4.5 Common blocks

F2PY generates wrappers to `common` blocks defined in a routine signature block. Common blocks are visible by all Fortran codes linked with the current extension module, but not to other extension modules (this restriction is due to how Python imports shared libraries). In Python, the F2PY wrappers to `common` blocks are `fortran` type objects that have (dynamic) attributes related to data members of common blocks. When accessed, these attributes return as Numeric array objects (multi-dimensional arrays are Fortran-contiguous) that directly link to data members in common blocks. Data members can be changed by direct assignment or by in-place changes to the corresponding array objects.

Example

Consider the following Fortran 77 code

```
C FILE: COMMON.F
      SUBROUTINE FOO
      INTEGER I,X
      REAL A
      COMMON /DATA/ I,X(4),A(2,3)
      PRINT*, "I=",I
      PRINT*, "X=[",X,"]"
      PRINT*, "A=["
      PRINT*, "[",A(1,1),",",A(1,2),",",A(1,3),"]"
      PRINT*, "[",A(2,1),",",A(2,2),",",A(2,3),"]"
      PRINT*, "]"
      END
C END OF COMMON.F
```

and wrap it using `f2py -c -m common common.f`.
In Python:

```
>>> import common
>>> print common.data.__doc__
i - 'i'-scalar
x - 'i'-array(4)
a - 'f'-array(2,3)

>>> common.data.i = 5
>>> common.data.x[1] = 2
>>> common.data.a = [[1,2,3],[4,5,6]]
>>> common.foo()
 I= 5
 X=[ 0 2 0 0]
 A=[
 [  1.,  2.,  3.]
 [  4.,  5.,  6.]
 ]
>>> common.data.a[1] = 45
>>> common.foo()
 I= 5
 X=[ 0 2 0 0]
 A=[
 [  1.,  2.,  3.]
 [  45.,  45.,  45.]
 ]
>>> common.data.a                 # a is Fortran-contiguous
array([[  1.,   2.,   3.],
       [ 45.,  45.,  45.]],'f')
```

4.6 Fortran 90 module data

The F2PY interface to Fortran 90 module data is similar to Fortran 77 common blocks.

Example

Consider the following Fortran 90 code

```
module mod
  integer i
  integer :: x(4)
```

```
  real, dimension(2,3) :: a
  real, allocatable, dimension(:,:) :: b
contains
  subroutine foo
    integer k
    print*, "i=",i
    print*, "x=[",x,"]"
    print*, "a=["
    print*, "[",a(1,1),",",a(1,2),",",a(1,3),"]"
    print*, "[",a(2,1),",",a(2,2),",",a(2,3),"]"
    print*, "]"
    print*, "Setting a(1,2)=a(1,2)+3"
    a(1,2) = a(1,2)+3
  end subroutine foo
end module mod
```

and wrap it using `f2py -c -m moddata moddata.f90`.
In Python:

```
>>> import moddata
>>> print moddata.mod.__doc__
i - 'i'-scalar
x - 'i'-array(4)
a - 'f'-array(2,3)
foo - Function signature:
  foo()


>>> moddata.mod.i = 5
>>> moddata.mod.x[:2] = [1,2]
>>> moddata.mod.a = [[1,2,3],[4,5,6]]
>>> moddata.mod.foo()
 i=           5
 x=[           1           2           0           0 ]
 a=[
 [   1.000000     ,   2.000000     ,   3.000000     ]
 [   4.000000     ,   5.000000     ,   6.000000     ]
 ]
 Setting a(1,2)=a(1,2)+3
>>> moddata.mod.a               # a is Fortran-contiguous
array([[ 1.,  5.,  3.],
       [ 4.,  5.,  6.]],'f')
```

4.6.1 Allocatable arrays

F2PY has basic support for Fortran 90 module allocatable arrays.

Example

Consider the following Fortran 90 code

```
module mod
  real, allocatable, dimension(:,:) :: b
contains
  subroutine foo
    integer k
    if (allocated(b)) then
       print*, "b=["
       do k = 1,size(b,1)
          print*, b(k,1:size(b,2))
```

```
      enddo
      print*, "]"
    else
      print*, "b is not allocated"
    endif
  end subroutine foo
end module mod
```

and wrap it using `f2py -c -m allocarr allocarr.f90`.
In Python:

```
>>> import allocarr
>>> print allocarr.mod.__doc__
b - 'f'-array(-1,-1), not allocated
foo - Function signature:
  foo()

>>> allocarr.mod.foo()
 b is not allocated
>>> allocarr.mod.b = [[1,2,3],[4,5,6]]         # allocate/initialize b
>>> allocarr.mod.foo()
 b=[
   1.000000       2.000000       3.000000
   4.000000       5.000000       6.000000
 ]
>>> allocarr.mod.b                             # b is Fortran-contiguous
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]],'f')
>>> allocarr.mod.b = [[1,2,3],[4,5,6],[7,8,9]] # reallocate/initialize b
>>> allocarr.mod.foo()
 b=[
   1.000000       2.000000       3.000000
   4.000000       5.000000       6.000000
   7.000000       8.000000       9.000000
 ]
>>> allocarr.mod.b = None                      # deallocate array
>>> allocarr.mod.foo()
 b is not allocated
```

5 Using F2PY

F2PY can be used either as a command line tool `f2py` or as a Python module `f2py2e`.

5.1 Command `f2py`

When used as a command line tool, `f2py` has three major modes, distinguished by the usage of `-c` and `-h` switches:

1. To scan Fortran sources and generate a signature file, use

   ```
   f2py -h <filename.pyf> <options> <fortran files>   \
     [[ only: <fortran functions>  : ]                \
      [ skip: <fortran functions>  : ]]...            \
     [<fortran files> ...]
   ```

   Note that a Fortran source file can contain many routines, and not necessarily all routines are needed to be used from Python. So, you can either specify which routines should be wrapped (in `only: .. :` part) or which routines F2PY should ignored (in `skip: .. :` part).

   If `<filename.pyf>` is specified as `stdout` then signatures are send to standard output instead of a file.

Among other options (see below), the following options can be used in this mode:

`--overwrite-signature` Overwrite existing signature file.

2. To construct an extension module, use

```
f2py <options> <fortran files>           \
  [[ only: <fortran functions>  : ]      \
   [ skip: <fortran functions>  : ]]... \
  [<fortran files> ...]
```

The constructed extension module is saved as `<modulename>module.c` to the current directory.

Here `<fortran files>` may also contain signature files. Among other options (see below), the following options can be used in this mode:

`--debug-capi` Add debugging hooks to the extension module. When using this extension module, various information about the wrapper is printed to standard output, for example, the values of variables, the steps taken, etc.

`-include'<includefile>'` Add a CPP `#include` statement to the extension module source. `<includefile>` should be given in one of the following forms:

```
"filename.ext"
<filename.ext>
```

The include statement is inserted just before the wrapper functions. This feature enables using arbitrary C functions (defined in `<includefile>`) in F2PY generated wrappers.

This option is deprecated. Use `usercode` statement to specify C codelets directly in signature filess

`--[no-]wrap-functions`

Create Fortran subroutine wrappers to Fortran functions. `--wrap-functions` is default because it ensures maximum portability and compiler independence.

`--include-paths <path1>:<path2>:..` Search include files from given directories.

`--help-link [<list of resources names>]` List system resources found by `scipy_distutils/system_info.py`. For example, try `f2py --help-link lapack_opt`.

3. To build an extension module, use

```
f2py -c <options> <fortran files>        \
  [[ only: <fortran functions>  : ]      \
   [ skip: <fortran functions>  : ]]... \
  [ <fortran/c source files> ] [ <.o, .a, .so files> ]
```

If `<fortran files>` contains a signature file, then a source for an extension module is constructed, all Fortran and C sources are compiled, and finally all object and library files are linked to the extension module `<modulename>.so` which is saved into the current directory.

If `<fortran files>` does not contain a signature file, then an extension module is constructed by scanning all Fortran source codes for routine signatures.

Among other options (see below) and options described in previous mode, the following options can be used in this mode:

`--help-fcompiler` List available Fortran compilers.

`--help-compiler` [**depreciated**]
List available Fortran compilers.

`--fcompiler=<Vendor>` Specify Fortran compiler type by vendor.

`--f77exec=<path>` Specify the path to F77 compiler

`--fcompiler-exec=<path>` [**depreciated**]
Specify the path to F77 compiler

`--f90exec=<path>` Specify the path to F90 compiler

`--f90compiler-exec=<path>` [**depreciated**]
Specify the path to F90 compiler

`--f77flags=<string>` Specify F77 compiler flags

`--f90flags=<string>` Specify F90 compiler flags

`--opt=<string>` Specify optimization flags

`--arch=<string>` Specify architecture specific optimization flags

`--noopt` Compile without optimization

`--noarch` Compile without arch-dependent optimization

`--debug` Compile with debugging information

`-l<libname>` Use the library `<libname>` when linking.

`-D<macro>[=<defn=1>]` Define macro `<macro>` as `<defn>`.

`-U<macro>` Define macro `<macro>`

`-I<dir>` Append directory `<dir>` to the list of directories searched for include files.

`-L<dir>` Add directory `<dir>` to the list of directories to be searched for `-l`.

`link-<resource>`

Link extension module with <resource> as defined by `scipy_distutils/system_info.py`. E.g. to link with optimized LAPACK libraries (vecLib on MacOSX, ATLAS elsewhere), use `--link-lapack_opt`. See also `--help-link` switch.

When building an extension module, a combination of the following macros may be required for non-gcc Fortran compilers:

```
-DPREPEND_FORTRAN
-DNO_APPEND_FORTRAN
-DUPPERCASE_FORTRAN
```

To test the performance of F2PY generated interfaces, use `-DF2PY_REPORT_ATEXIT`. Then a report of various timings is printed out at the exit of Python. This feature may not work on all platforms, currently only Linux platform is supported.

To see whether F2PY generated interface performs copies of array arguments, use `-DF2PY_REPORT_ON_ARRAY_COPY=<int>`. When the size of an array argument is larger than `<int>`, a message about the coping is sent to `stderr`.

Other options:

`-m <modulename>` Name of an extension module. Default is `untitled`.

`--[no-]lower` Do [not] lower the cases in `<fortran files>`. By default, `--lower` is assumed with `-h` switch, and `--no-lower` without the `-h` switch.

`--build-dir <dirname>` All F2PY generated files are created in `<dirname>`. Default is `tempfile.mktemp()`.

`--quiet` Run quietly.

`--verbose` Run with extra verbosity.

`-v` Print f2py version ID and exit.

Execute `f2py` without any options to get an up-to-date list of available options.

5.2 Python module `f2py2e`

Warning

The current Python interface to `f2py2e` module is not mature and may change in future depending on users needs.

The following functions are provided by the `f2py2e` module:

`run_main(<list>)` Equivalent to running:

```
f2py <args>
```

where `<args>=string.join(<list>,' ')`, but in Python. Unless `-h` is used, this function returns a dictionary containing information on generated modules and their dependencies on source files. For example, the command `f2py -m scalar scalar.f` can be executed from Python as follows

```
>>> import f2py2e
>>> r=f2py2e.run_main(['-m','scalar','docs/usersguide/scalar.f'])
Reading fortran codes...
        Reading file 'docs/usersguide/scalar.f'
Post-processing...
        Block: scalar
                        Block: FOO
Building modules...
        Building module "scalar"...
        Wrote C/API module "scalar" to file "./scalarmodule.c"
>>> print r
{'scalar': {'h': ['/home/users/pearu/src_cvs/f2py2e/src/fortranobject.h'],
  'csrc': ['./scalarmodule.c',
                  '/home/users/pearu/src_cvs/f2py2e/src/fortranobject.c']}}
```

You cannot build extension modules with this function, that is, using `-c` is not allowed. Use `compile` command instead, see below.

`compile(source, modulename='untitled', extra_args='', verbose=1, source_fn=None)`

Build extension module from Fortran 77 source string `source`. Return 0 if successful. Note that this function actually calls `f2py -c ..` from shell to ensure safety of the current Python process. For example,

```
>>> import f2py2e
>>> fsource = '''
...       subroutine foo
...       print*, "Hello world!"
...       end
... '''
>>> f2py2e.compile(fsource,modulename='hello',verbose=0)
0
>>> import hello
>>> hello.foo()
 Hello world!
```

6 Using `scipy_distutils`

`scipy_distutils` is part of the SciPy project and aims to extend standard Python `distutils` to deal with Fortran sources and F2PY signature files, e.g. compile Fortran sources, call F2PY to construct extension modules, etc.

Example

Consider the following setup file:

```
#!/usr/bin/env python
# File: setup_example.py

from scipy_distutils.core import Extension

ext1 = Extension(name = 'scalar',
                 sources = ['scalar.f'])
```

```
ext2 = Extension(name = 'fib2',
                 sources = ['fib2.pyf','fib1.f'])

if __name__ == "__main__":
    from scipy_distutils.core import setup
    setup(name = 'f2py_example',
          description       = "F2PY Users Guide examples",
          author            = "Pearu Peterson",
          author_email      = "pearu@cens.ioc.ee",
          ext_modules = [ext1,ext2]
          )
# End of setup_example.py
```

Running

```
python setup_example.py build
```

will build two extension modules `scalar` and `fib2` to the build directory.
`scipy_distutils` extends `distutils` with the following features:

- `Extension` class argument `sources` may contain Fortran source files. In addition, the list `sources` may contain at most one F2PY signature file, and then the name of an Extension module must match with the `<modulename>` used in signature file. It is assumed that an F2PY signature file contains exactly one `python module` block.

  If `sources` does not contain a signature files, then F2PY is used to scan Fortran source files for routine signatures to construct the wrappers to Fortran codes.

  Additional options to F2PY process can be given using `Extension` class argument `f2py_options`.

6.1 `scipy_distutils` 0.2.2 and up

- The following new `distutils` commands are defined:

  `build_src` to construct Fortran wrapper extension modules, among many other things.

  `config_fc` to change Fortran compiler options

  as well as `build_ext` and `build_clib` commands are enhanced to support Fortran sources.

  Run

  ```
  python <setup.py file> config_fc build_src build_ext --help
  ```

  to see available options for these commands.

- When building Python packages containing Fortran sources, then one can choose different Fortran compilers by using `build_ext` command option `--fcompiler=<Vendor>`. Here `<Vendor>` can be one of the following names:

  ```
  absoft sun mips intel intelv intele intelev nag compaq compaqv gnu vast pg hpux
  ```

  See `scipy_distutils/fcompiler.py` for up-to-date list of supported compilers or run

  ```
  f2py -c --help-fcompiler
  ```

6.2 `scipy_distutils` pre 0.2.2

- The following new `distutils` commands are defined:

  `build_flib` to build f77/f90 libraries used by Python extensions;

  `run_f2py` to construct Fortran wrapper extension modules.

  Run

```
python <setup.py file> build_flib run_f2py --help
```

to see available options for these commands.

- When building Python packages containing Fortran sources, then one can choose different Fortran compilers either by using `build_flib` command option `--fcompiler=<Vendor>` or by defining environment variable `FC_VENDOR=<Vendor>`. Here `<Vendor>` can be one of the following names:

  ```
  Absoft Sun SGI Intel Itanium NAG Compaq Digital Gnu VAST PG
  ```

  See `scipy_distutils/command/build_flib.py` for up-to-date list of supported compilers.

7 Extended F2PY usages

7.1 Adding self-written functions to F2PY generated modules

Self-written Python C/API functions can be defined inside signature files using `usercode` and `pymethoddef` statements (they must be used inside the `python module` block). For example, the following signature file `spam.pyf`

```
!    -*- f90 -*-
python module spam
    usercode '''
  static char doc_spam_system[] = "Execute a shell command.";
  static PyObject *spam_system(PyObject *self, PyObject *args)
  {
    char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
  }
    '''
    pymethoddef '''
    {"system",  spam_system, METH_VARARGS, doc_spam_system},
    '''
end python module spam
```

wraps the C library function `system()`:

```
f2py -c spam.pyf
```

In Python:

```
>>> import spam
>>> status = spam.system('whoami')
pearu
>> status = spam.system('blah')
sh: line 1: blah: command not found
```

7.2 Modifying the dictionary of a F2PY generated module

The following example illustrates how to add an user-defined variables to a F2PY generated extension module. Given the following signature file

```
!    -*- f90 -*-
python module var
  usercode '''
    int BAR = 5;
```

```
  '''
  interface
    usercode '''
      PyDict_SetItemString(d,"BAR",PyInt_FromLong(BAR));
    '''
  end interface
end python module
```

compile it as `f2py -c var.pyf`.

Notice that the second `usercode` statement must be defined inside an `interface` block and where the module dictionary is available through the variable `d` (see `f2py var.pyf`-generated `varmodule.c` for additional details).

In Python:

```
>>> import var
>>> var.BAR
5
```

Generated on: 2005-01-30 18:58 UTC. Generated by Docutils from reStructuredText source.