f2py
Fortran to Python Interface Generator

Second Edition

Pearu Peterson <pearu@ioc.ee>

Revision : 1.16
December 12, 2013

Abstract

f2py is a Python program that generates Python C/API modules for wrapping Fortran 77/90/95 codes to Python. The user can influence the process by modifying the signature files that f2py generates when scanning the Fortran codes. This document describes the syntax of the signature files and the ways how the user can dictate the tool to produce wrapper functions with desired Python signatures. Also how to call the wrapper functions from Python is discussed.

See http://cens.ioc.ee/projects/f2py2e/ for updates of this document and the tool.

Contents

1 Introduction

`f2py` is a command line tool that generates Python C/API modules for interfacing Fortran 77/90/95 codes and Fortran 90/95 modules from Python. In general, using `f2py` an interface is produced in three steps:

(i) `f2py` scans Fortran sources and creates the so-called *signature* file; the signature file contains the signatures of Fortran routines; the signatures are given in the free format of the Fortran 90/95 language specification. Latest version of `f2py` generates also a make file for building shared module. About currently supported compilers see the `f2py` home page

(ii) Optionally, the signature files can be modified manually in order to dictate how the Fortran routines should be called or seemed from the Python environment.

(iii) `f2py` reads the signature files and generates Python C/API modules that can be compiled and imported to Python code. In addition, a LaTeX document is generated that contains the documentation of wrapped functions.

(Note that if you are satisfied with the default signature that `f2py` generates in step (i), all three steps can be covered with just one call to `f2py`— by not specifying '`-h`' flag). Latest versions of `f2py` support so-called `f2py` directive that allows inserting various information about wrapping directly to Fortran source code as comments (`<comment char>f2py <signature statement>`).

The following diagram illustrates the usage of the tool:

```
! Fortran file foo.f:
      subroutine foo(a)
      integer a
      a = a + 5
      end

! Fortran file bar.f:
      function bar(a,b)
      integer a,b,bar
      bar = a + b
      end
```

(i) `sh> f2py foo.f bar.f -m foobar -h foobar.pyf`

```
!%f90
! Signature file: foobar.pyf
python module foobar ! in
    interface  ! in :foobar
        subroutine foo(a) ! in :foobar:foo.f
```

```
            integer intent(inout) :: a
        end subroutine foo
        function bar(a,b) ! in :foobar:bar.f
            integer :: a
            integer :: b
            integer :: bar
        end function bar
    end interface
end python module foobar
```

(ii) Edit the signature file (here I made `foo`s argument `a` to be `intent(inout)`, see Sec. 2.3).

(iii) `sh> f2py foobar.pyf`

```
/* Python C/API module: foobarmodule.c */
...
```

(iv) `sh> make -f Makefile-foobar`

```
Python shared module: foobarmodule.so
```

(v) Usage in Python:

```
>>> import foobar
>>> print foobar.__doc__
This module 'foobar' is auto-generated with f2py (version:1.174).
The following functions are available:
  foo(a)
  bar = bar(a,b)
.
>>> print foobar.bar(2,3)
5
>>> from Numeric import *
>>> a = array(3)
>>> print a,foobar.foo(a),a
3 None 8
```

Information about how to call `f2py` (steps (i) and (iii)) can be obtained by executing

```
  sh> f2py
```

This will print the usage instructions. Step (iv) is system dependent (compiler and the locations of the header files `Python.h` and `arrayobject.h`), and so you must know how to compile a shared module for Python in you system.

The next Section describes the step (ii) in more detail in order to explain how you can influence to the process of interface generation so that the users can enjoy more writing Python programs using your wrappers that call Fortran routines. Step (v) is covered in Sec. 3.

1.1 Features

`f2py` has the following features:

1. `f2py` scans real Fortran codes and produces the signature files. The syntax of the signature files is borrowed from the Fortran 90/95 language specification with some extensions.

2. `f2py` uses the signature files to produce the wrappers for Fortran 77 routines and their `COMMON` blocks.

3. For `external` arguments `f2py` constructs a very flexible call-back mechanism so that Python functions can be called from Fortran.

4. You can pass in almost arbitrary Python objects to wrapper functions. If needed, `f2py` takes care of type-casting and non-contiguous arrays.

5. You can modify the signature files so that `f2py` will generate wrapper functions with desired signatures. `depend()` attribute is introduced to control the initialization order of the variables. `f2py` introduces `intent(hide)` attribute to remove the particular argument from the argument list of the wrapper function. In addition, `optional` and `required` attributes are introduced and employed.

6. `f2py` supports almost all standard Fortran 77/90/95 constructs and understands all basic Fortran types, including (multi-dimensional, complex) arrays and character strings with adjustable and assumed sizes/lengths.

7. `f2py` generates a LaTeX document containing the documentations of the wrapped functions (argument types, dimensions, etc). The user can easily add some human readable text to the documentation by inserting `note(<LaTeX text>)` attribute to the definition of routine signatures.

8. `f2py` generates a GNU make file that can be used for building shared modules calling Fortran functions.

9. `f2py` supports wrapping Fortran 90/95 module routines.

2 Signature file

The syntax of a signature file is borrowed from the Fortran 90/95 language specification. Almost all Fortran 90/95 standard constructs are understood. Recall that Fortran 77 is a subset of Fortran 90/95. This tool introduces also some new attributes that are used for controlling the process of Fortran to Python interface construction. In the following, a short overview of the constructs used in signature files will be given.

2.1 Module block

A signature file contains one or more `pythonmodule` blocks. A `pythonmodule` block has the following structure:

```
python module <modulename>
  interface
    <routine signatures>
  end [interface]
  interface
    module <F90/95 modulename>
      <F90 module data type declarations>
      <F90 module routine signatures>
    end [module [<F90/95 modulename>]]
  end [interface]
end [pythonmodule [<modulename>]]
```

For each `pythonmodule` block `f2py` will generate a C-file `<modulename>module.c` (see step (iii)). (This is not true if `<modulename>` contains substring `__user__`, see Sec. 2.7 and `external` attribute).

2.2 Signatures of Fortran routines and Python functions

The signature of a Fortran routine has the following structure:

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

Let us introduce also the signature of the corresponding wrapper function:

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

Before you edit the signature file, you should first decide what is the desired signature of the corresponding Python function. `f2py` offers many possibilities to control the interface construction process: you may want to insert/change/remove various attributes in the declarations of the arguments in order to change the appearance of the arguments in the Python wrapper function.

- The definition of the `<argument type declaration>` 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>)
  ```

  (there is no sense to modify `<typespec>`s generated by `f2py`). `<attrspec>` is a comma separated list of attributes (see Sec. 2.3);

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

  where `<arrayspec>` is a comma separated list of dimension bounds; `<init_expr>` is a C-expression (see Sec. 2.4). If an argument is not defined with `<argument type declaration>`, its type is determined by applying `implicit` rules (if it is not specifyied, then standard rules are applied).

- The definition of the `<argument attribute statement>` is a short form of the `<argument type declaration>`:

  ```
  <attrspec> <entitydecl>
  ```

- `<use statement>` is defined as follows

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

Currently the use statement is used to link call-back modules (Sec. 2.7) and the external arguments (call-back functions).

- <common block statement> is defined as follows

  ```
  common /<commonname>/ <shortentitydecl>
  ```

  where

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

  One module block should not contain two or more common blocks with the same name. Otherwise, the later ones are ignored. The types of variables in <shortentitydecl> can be defined in <argument type declarations>. Note that there you can specify also the array specifications; then you don't need to do that in <shortentitydecl>.

2.3 Attributes

The following attributes are used by f2py:

optional — the variable is moved to the end of optional argument list of the wrapper function. Default value of an optional argument can be specified using <init_expr> in entitydecl. You can use optional attribute also for external arguments (call-back functions), but it is your responsibility to ensure that it is given by the user if Fortran routine wants to call it.

required — the variable is considered as a required argument (that is default). You will need this in order to overwrite the optional attribute that is automatically set when <init_expr> is used. However, usage of this attribute should be rare.

dimension(<arrayspec>) — used when the variable is an array. For unbounded dimensions symbols '*' or ':' can be used (then internally the corresponding dimensions are set to -1; you'll notice this when certain exceptions are raised).

external — the variable is a call-back function. f2py will construct a call-back mechanism for this function. Also call-back functions must be defined by their signatures, and there are several ways to do that. In most cases, f2py will be able to determine the signatures of call-back functions from the Fortran source code; then it builds an additional module block with a name containing string '__user__' (see Sec. 2.7) and includes use statement to the routines signature. Anyway, you should check that the generated signature is correct.

Alternatively, you can specify the signature by inserting to the routines block a "model" how the call-back function would be called from Fortran. For subroutines you should use

```
call <call-back name>(<arguments>)
```

and for functions

```
<return value> = <call-back name>(<arguments>)
```

The variables in <arguments> and <return value> must be defined as well. You can use the arguments of the main routine, for instance.

intent(<intentspec>) — this specifies the "intention" of the variable. <intentspec> is a comma separated list of the following specifications:

in — the variable is considered to be an input variable (default). It means that the Fortran function uses only the value(s) of the variable and is assumed not to change it.

inout — the variable is considered to be an input/output variable which means that Fortran routine may change the value(s) of the variable. Note that in Python only array objects can be changed "in place". (intent(outin) is intent(inout).)

`out` — the value of the (output) variable is returned by the wrapper function: it is appended to the list of `<returned variables>`. If `out` is specified alone, also `hide` is assumed.

`hide` — use this if the variable *should not* or *need not* to be in the list of wrapper function arguments (not even in optional ones). For example, this is assumed if `intent(out)` is used. You can "hide" an argument if it has always a constant value specified in `<init_expr>`, for instance.

The following rules apply:

- if no `intent` attribute is specified, `intent(in)` is assumed;
- `intent(in,inout)` is `intent(in)`;
- `intent(in,hide)`, `intent(inout,hide)` are `intent(hide)`;
- `intent(out)` is `intent(out,hide)`;
- `intent(inout)` is NOT `intent(in,out)`.

In conclusion, the following combinations are "minimal": `intent(in)`, `intent(inout)`, `intent(out)`, `intent(hide)`, `intent(in,out)`, and `intent(inout,out)`.

`check([<C-booleanexpr>])` — if `<C-booleanexpr>` evaluates to zero, an exception is raised about incorrect value or size or any other incorrectness of the variable. If `check()` or `check` is used then `f2py` will not try to guess the checks automatically.

`depend([<names>])` — the variable depends on other variables listed in `<names>`. These dependence relations determine the order of internal initialization of the variables. If you need to change these relations then be careful not to break the dependence relations of other relevant variables. If `depend()` or `depend` is used then `f2py` will not try to guess the dependence relations automatically.

`note(<LaTeX text>)` — with this attribute you can include human readable documentation strings to the LaTeX document that `f2py` generates. Do not insert here information that `f2py` can establish by itself, such as, types, sizes, lengths of the variables. Here you can insert almost arbitrary LaTeX text. Note that `<LaTeX text>` is mainly used inside the LaTeX `description` environment. Hint: you can use `\texttt{<name>}` for typesetting variable `<name>` in LaTeX. In order to get a new line to the LaTeX document, use `\n` followed by a space. For longer text, you may want to use line continuation feature of Fortran 90/95 language: set `&` (ampersand) to be the last character in a line.

`parameter` — the variable is parameter and it must have a value. If the parameter is used in dimension specification, it is replaced by its value. (Are there any other usages of parameters except in dimension specifications? Let me know and I'll add support for it).

2.4 C-expressions

The signature of a routine may contain C-expressions in

- `<init_expr>` for initializing particular variable, or in
- `<C-booleanexpr>` of the `check` attribute, or in
- `<arrayspec>` of the `dimension` attribute.

A C-expression may contain

- standard C-statement,
- functions offered in `math.h`,

- previously initialized variables (study the dependence relations) from the argument list, and
- the following CPP-macros:

  `len(<name>)` — the length of an array `<name>`;

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

  `rank(<name>)` — the rank of an array `<name>`;

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

In addition, when initializing arrays, an index vector `int _i[rank(<name>)];` is available: `_i[0]` refers to the index of the first dimension, `_i[1]` to the index of the second dimension, etc. For example, the argument type declaration

```
integer a(10) = _i[0]
```

is equivalent with the following Python statement

```
a = array(range(10))
```

2.5 Required/optional arguments

When `optional` attribute is used (including the usage of `<init_expr>` without the `required` attribute), the corresponding variable in the argument list of a Fortran routine is appended to the optional argument list of the wrapper function.

For optional array argument all dimensions must be bounded (not `(*)` or `(:)`) and defined at the time of initialization (dependence relations).

If the `None` object is passed in in place of a required array argument, it will be considered as optional: that is, the memory is allocated (of course, if it has unbounded dimensions, an exception will be raised), and if `<init_expr>` is defined, initialization is carried out.

2.6 Internal checks

All array arguments are checked against the correctness of their rank. If there is a mismatch, `f2py` attempts to fix that by constructing an array with a correct rank from the given array argument (there will be no performance hit as no data is copied). The freedom to do so is given only if some dimensions are unbounded or their value is 1. An exception is raised when the sizes will not match.

All bounded dimensions of an array are checked to be larger or equal to the dimensions specified in the signature.

So, you don't need to give explicit `check` attributes to check these internal checks.

2.7 Call-back modules

A Fortran routine may have `external` arguments (call-back functions). The signatures of the call-back functions must be defined in a call-back `module` block (its name contains `__user__`), in general; other possibilities are described in the `external` attribute specification (see Sec. 2.3). For the signatures of call-back functions the following restrictions apply:

- Attributes `external`, `check(...)`, and initialization statements are ignored.
- Attribute `optional` is used only for changing the order of the arguments.
- For arrays all dimension bounds must be specified. They may be C-expressions containing variables from the argument list. Note that here CPP-macros `len`, `shape`, `rank`, and `slen` are not available.

2.8 Common blocks

All fields in a common block are mapped to arrays of appropriate sizes and types. Scalars are mapped to rank-0 arrays. For multi-dimensional fields the corresponding arrays are transposed. In the type declarations of the variables representing the common block fields, only `dimension(<arrayspec>)`, `intent(hide)`, and `note(<LaTeX text>)` attributes are used, others are ignored.

2.9 Including files

You can include files to the signature file using

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

statement. It can be used in any part of the signature file. If the file `<filename>` does not exists or it is not in the path, the `include` line is ignored.

2.10 `f2py` directives

You can insert signature statements directly to Fortran source codes as comments. Anything that follows `<comment char>f2py` is regarded as normal statement for `f2py`.

3 Calling wrapper functions from Python

3.1 Scalar arguments

In general, for scalar argument you can pass in in addition to ordinary Python scalars (like integers, floats, complex values) also arbitrary sequence objects (lists, arrays, strings) — then the first element of a sequence is passed in to the Fortran routine.

It is recommended that you always pass in scalars of required type. This ensures the correctness as no type-casting is needed. However, no exception is raised if type-casting would produce inaccurate or incorrect results! For example, in place of an expected complex value you can give an integer, or vice-versa (in the latter case only a rounded real part of the complex value will be used).

If the argument is `intent(inout)` then Fortran routine can change the value "in place" only if you pass in a sequence object, for instance, rank-0 array. Also make sure that the type of an array is of correct type. Otherwise type-casting will be performed and you may get inaccurate or incorrect results. The following example illustrates this

```
>>> a = array(0)
>>> calculate_pi(a)
>>> print a
3
```

If you pass in an ordinary Python scalar in place of `intent(inout)` variable, it will be used as an input argument since Python scalars cannot not be changed "in place" (all Python scalars are immutable objects).

3.2 String arguments

You can pass in strings of arbitrary length. If the length is greater than required, only a required part of the string is used. If the length is smaller than required, additional memory is allocated and fulfilled with '`\0`'s.

Because Python strings are immutable, `intent(inout)` argument expects an array version of a string — an array of chars: `array("<string>")`. Otherwise, the change "in place" has no effect.

3.3 Array arguments

If the size of an array is relatively large, it is *highly recommended* that you pass in arrays of required type. Otherwise, type-casting will be performed which includes the creation of new arrays and their copying. If the argument is also `intent(inout)`, the wasted time is doubled. So, pass in arrays of required type!

On the other hand, there are situations where it is perfectly all right to ignore this recommendation: if the size of an array is relatively small or the actual time spent in Fortran routine takes much longer than copying an array. Anyway, if you want to optimize your Python code, start using arrays of required types.

Another source of performance hit is when you use non-contiguous arrays. The performance hit will be exactly the same as when using incorrect array types. This is because a contiguous copy is created to be passed in to the Fortran routine.

`f2py` provides a feature such that the ranks of array arguments need not to match — only the correct total size matters. For example, if the wrapper function expects a rank-1 array `array([...])`, then it is correct to pass in rank-2 (or higher) arrays `array([[...],...,[...]])` assuming that the sizes will match. This is especially useful when the arrays should contain only one element (size is 1). Then you can pass in arrays `array(0)`, `array([0])`, `array([[0]])`, etc and all cases are handled correctly. In this case it is correct to pass in a Python scalar in place of an array (but then "change in place" is ignored, of course).

3.3.1 Multidimensional arrays

If you are using rank-2 or higher rank arrays, you must always remember that indexing in Fortran starts from the lowest dimension while in Python (and in C) the indexing starts from the highest dimension (though some compilers have switches to change this). As a result, if you pass in a 2-dimensional array then the Fortran routine sees it as the transposed version of the array (in multi-dimensional case the indexes are reversed).

You must take this matter into account also when modifying the signature file and interpreting the generated Python signatures:

- First, when initializing an array using `init_expr`, the index vector `_i[]` changes accordingly to Fortran convention.
- Second, the result of CPP-macro `shape(<array>,0)` corresponds to the last dimension of the Fortran array, etc.

Let me illustrate this with the following example:

```
! Fortran file: arr.f
       subroutine arr(l,m,n,a)
       integer l,m,n
       real*8 a(l,m,n)
       ...
       end
```

`f2py` will generate the following signature file:

```
!%f90
! Signature file: arr.f90
python module arr ! in
  interface  ! in :arr
    subroutine arr(l,m,n,a) ! in :arr:arr.f
      integer optional,check(shape(a,2)==l),depend(a) :: l=shape(a,2)
```

```
      integer optional,check(shape(a,1)==m),depend(a) :: m=shape(a,1)
      integer optional,check(shape(a,0)==n),depend(a) :: n=shape(a,0)
      real*8 dimension(l,m,n) :: a
    end subroutine arr
  end interface
end python module arr
```

and the following wrapper function will be produced

```
None = arr(a,l=shape(a,2),m=shape(a,1),n=shape(a,0))
```

In general, I would suggest not to specify the given optional variables `l,m,n` when calling the wrapper function — let the interface find the values of the variables `l,m,n`. But there are occasions when you need to specify the dimensions in Python.

So, in Python a proper way to create an array from the given dimensions is

```
>>> a = zeros(n,m,l,'d')
```

(note that the dimensions are reversed and correct type is specified), and then a complete call to `arr` is

```
>>> arr(a,l,m,n)
```

From the performance point of view, always be consistent with Fortran indexing convention, that is, use transposed arrays. But if you do the following

```
>>> a = transpose(zeros(l,m,n,'d'))
>>> arr(a)
```

then you will get a performance hit! The reason is that here the transposition is not actually performed. Instead, the array `a` will be non-contiguous which means that before calling a Fortran routine, internally a contiguous array is created which includes memory allocation and copying. In addition, if the argument array is also `intent(inout)`, the results are copied back to the initial array which doubles the performance hit!

So, to improve the performance: always pass in arrays that are contiguous.

3.3.2 Work arrays

Often Fortran routines use the so-called work arrays. The corresponding arguments can be declared as optional arguments, but be sure that all dimensions are specified (bounded) and defined before the initialization (dependence relations).

On the other hand, if you call the Fortran routine many times then you don't want to allocate/deallocate the memory of the work arrays on every call. In this case it is recommended that you create temporary arrays with proper sizes in Python and use them as work arrays. But be careful when specifying the required type and be sure that the temporary arrays are contiguous. Otherwise the performance hit would be even harder than the hit when not using the temporary arrays from Python!

3.4 Call-back arguments

`f2py` builds a very flexible call-back mechanisms for call-back arguments. If the wrapper function expects a call-back function `fun` with the following Python signature to be passed in

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

but the user passes in a function `gun` with the signature

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

and the following extra arguments (specified as additional optional argument for the wrapper function):

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

then the actual call-back is constructed accordingly to the following rules:

- if p==0 then gun(a_1,...,a_q), where q=min(m,n);
- if n+p<=m then gun(a_1,...,a_n,e_1,...,e_p);
- if p<=m<n+p then gun(a_1,...,a_q,e_1,...,e_p), where q=m-p;
- if p>m then gun(e_1,...,e_m);
- if n+p is less than the number of required arguments of the function gun, an exception is raised.

A call-back function gun may return any number of objects as a tuple: if k<l, then objects y_k+1,...,y_l are ignored; if k>l, then only objects x_1,...,x_l are set.

3.5 Obtaining information on wrapper functions

From the previous sections we learned that it is useful for the performance to pass in arguments of expected type, if possible. To know what are the expected types, f2py generates a complete documentation strings for all wrapper functions. You can read them from Python by printing out __doc__ attributes of the wrapper functions. For the example in Sec. 1:

```
>>> print foobar.foo.__doc__
Function signature:
  foo(a)
Required arguments:
  a : in/output rank-0 array(int,'i')
>>> print foobar.bar.__doc__
Function signature:
  bar = bar(a,b)
Required arguments:
  a : input int
  b : input int
Return objects:
  bar : int
```

In addition, f2py generates a LaTeX document (<modulename>module.tex) containing a bit more information on the wrapper functions. See for example Appendix that contains a result of the documentation generation for the example module foobar. Here the file foobar-smart.f90 (modified version of foobar.f90) is used — it contains note(<LaTeX text>) attributes for specifying some additional information.

3.6 Wrappers for common blocks

[See examples test-site/e/runme*]

What follows is obsolute for f2py version higher that 2.264.

f2py generates wrapper functions for common blocks. For every common block with a name <commonname> a function get_<commonname>() is constructed that takes no arguments and returns a dictionary. The dictionary represents maps between the names of common block fields and the arrays containing the common block fields (multi-dimensional arrays are transposed). So, in order to access to the common block fields, you must first obtain the references

```
commonblock = get_<commonname>()
```

and then the fields are available through the arrays `commonblock["<fieldname>"]`. To change the values of common block fields, you can use for scalars

```
commonblock["<fieldname>"][0] = <new value>
```

and for arrays

```
commonblock["<fieldname>"][:] = <new array>
```

for example.

For more information on the particular common block wrapping, see `get_<commonname>.__doc__`.

3.7 Wrappers for F90/95 module data and routines

[See example `test-site/mod/runme_mod`]

3.8 Examples

Examples on various aspects of wrapping Fortran routines to Python can be found in directories `test-site/d/` and `test-site/e/`: study the shell scripts `runme_*`. See also files in `doc/ex1/`.

4 f2py command line options

`f2py` has the following command line syntax (run `f2py` without arguments to get up to date options!!!):

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

where

`<options>` — the following options are available:

- `-f77` — `<fortran files>` are in Fortran 77 fixed format (default).
- `-f90` — `<fortran files>` are in Fortran 90/95 free format (default for signature files).
- `-fix` — `<fortran files>` are in Fortran 90/95 fixed format.
- `-h <filename>` — after scanning the `<fortran files>` write the signatures of Fortran routines to file `<filename>` and exit. If `<filename>` exists, `f2py` quits without overwriting the file. Use `--overwrite-signature` to overwrite.
- `-m <modulename>` — specify the name of the module when scanning Fortran 77 codes for the first time. `f2py` will generate Python C/API module source `<modulename>module.c`.
- `--lower/--no-lower` — lower/do not lower the cases when scanning the `<fortran files>`. Default when `-h` flag is specified/unspecified (that is for Fortran 77 codes/signature files).
- `--short-latex` — use this flag when you want to include the generated LaTeX document to another LaTeX document.
- `--debug-capi` — create a very verbose C/API code. Useful for debbuging.
- `-makefile <options>` — run `f2py` without arguments for more information.

--use-libs — see -makefile.

--overwrite-makefile — overwrite existing Makefile-<modulename>.

-v — print f2py version number and exit.

-pyinc — print Python include path and exit.

<fortran files> — are the paths to Fortran files or to signature files that will be scanned for <fortran functions> in order to determine their signatures.

<fortran functons> — are the names of Fortran routines for which Python C/API wrapper functions will be generated. Default is all that are found in <fortran files>.

only:/skip: — are flags for filtering in/out the names of fortran routines to be wrapped. Run f2py without arguments for more information about the usage of these flags.

5 Bugs, Plans, and Feedback

Currently no bugs have found that I was not able to fix. I will be happy to receive bug reports from you (so that I could fix them and keep the first sentence of this paragraph as true as possible ;-). Note that f2py is developed to work properly with gcc/g77 compilers.

NOTE: Wrapping callback functions returning COMPLEX may fail on some systems. Workaround: avoid it by using callback subroutines.

Here follows a list of things that I plan to implement in (near) future:

1. recognize file types by their extension (signatures: *.pyf, Fortran 77, Fortran 90 fixed: *.f, *.for, *.F, *.FOR, Fortran 90 free: *.F90, *.f90, *.m, *.f95, *.F95); [DONE]
2. installation using distutils (when it will be stable);
3. put out to the web examples of f2py usages in real situations: wrapping vode, for example;
4. implement support for PARAMETER statement; [DONE]
5. rewrite test-site;
6. ...

and here are things that I plan to do in future:

1. implement intent(cache) attribute for an optional work arrays with a feature of allocating additional memory if needed;
2. use f2py for wrapping Fortran 90/95 codes. f2py should scan Fortran 90/95 codes with no problems, what needs to be done is find out how to call a Fortran 90/95 function (from a module) from C. Anybody there willing to test f2py with Fortran 90/95 modules? [DONE]
3. implement support for Fortran 90/95 module data; [DONE]
4. implement support for BLOCK DATA blocks (if needed);
5. test/document f2py for CHARACTER arrays;
6. decide whether internal transposition of multi-dimensional arrays is reasonable (need efficient code then), even if this is controlled by the user trough some additional keyword; need consistent and safe policy here;
7. use f2py for generating wrapper functions also for C programs (a kind of SWIG, only between Python and C). For that f2py needs a command line switch to inform itself that C scalars are passed in by their value, not by their reference, for instance;

8. introduce a counter that counts the number of inefficient usages of wrapper functions (copying caused by type-casting, non-contiguous arrays);
9. if needed, make `DATA` statement to work properly for arrays;
10. rewrite `COMMON` wrapper; [DONE]
11. ...

I'll appreciate any feedback that will improve `f2py` (bug reports, suggestions, etc). If you find a correct Fortran code that fails with `f2py`, try to send me a minimal version of it so that I could track down the cause of the failure. Note also that there is no sense to send me files that are auto-generated with `f2py` (I can generate them myself); the version of `f2py` that you are using (run `f2py -v`), and the relevant fortran codes or modified signature files should be enough information to fix the bugs. Also add some information on compilers and linkers that you use to the bug report.

6 History of `f2py`

1. I was driven to start developing a tool such as `f2py` after I had wrote several Python C/API modules for interfacing various Fortran routines from the Netlib. This work was tedious (some of functions had more than 20 arguments, only few of them made sense for the problems that they solved). I realized that most of the writing could be done automatically.
2. On 9th of July, 1999, the first lines of the tool was written. A prototype of the tool was ready to use in only three weeks. During this time Travis Oliphant joined to the project and shared his valuable knowledge and experience; the call-back mechanism is his major contribution. Then I gave the tool to public under the name FPIG — *Fortran to Python Interface Generator*. The tool contained only one file `f2py.py`.
3. By autumn, it was clear that a better implementation was needed as the debugging process became very tedious. So, I reserved some time and rewrote the tool from scratch. The most important result of this rewriting was the code that reads real Fortran codes and determines the signatures of the Fortran routines. The main attention was payed in particular to this part so that the tool could read arbitrary Fortran 77/90/95 codes. As a result, the other side of the tools task, that is, generating Python C/API functions, was not so great. In public, this version of the tool was called `f2py2e` — *Fortran to Python C/API generator, the Second Edition.*
4. So, a month before The New Year 2000, I started the third iteration of the `f2py` development. Now the main attention was to have a good C/API module constructing code. By 21st of January, 2000, the tool of generating wrapper functions for Fortran routines was ready. It had many new features and was more robust than ever.
5. In 25th of January, 2000, the first public release of `f2py` was announced (version 1.116).
6. In 12th of September, 2000, the second public release of `f2py` was announced (version 2.264). It now has among other changes a support for Fortran 90/95 module routines.

A Module `foobar`

This module contains two examples that are used in `f2py` documentation.

A.1 Wrapper function foo

foo(a) — Example of a wrapper function of a Fortran subroutine.
Required arguments:

a : in/output rank-0 array(int,'i') — 5 is added to the variable a "in place".

A.2 Wrapper function bar

bar = bar(a, b) — Add two values.
Required arguments:

a : input int — The first value.

b : input int — The second value.

Return objects:

bar : int — See elsewhere.

B Applications

B.1 Example: wrapping C library fftw

Here follows a simple example how to use f2py to generate a wrapper for C functions. Let us create a FFT code using the functions in FFTW library. I'll assume that the library fftw is configured with --enable-shared option.

Here is the wrapper for the typical usage of FFTW:

```
/* File: wrap_dfftw.c */
#include <dfftw.h>

extern void dfftw_one(fftw_complex *in,fftw_complex *out,int *n) {
  fftw_plan p;
  p = fftw_create_plan(*n,FFTW_FORWARD,FFTW_ESTIMATE);
  fftw_one(p,in,out);
  fftw_destroy_plan(p);
}
```

and here follows the corresponding siganture file (created manually):

```
!%f90
! File: fftw.f90
module fftw
  interface
     subroutine dfftw_one(in,out,n)
       integer n
       complex*16 in(n),out(n)
       intent(out) out
       intent(hide) n
     end subroutine dfftw_one
  end interface
end module fftw
```

Now let us generate the Python C/API module with f2py:

```
f2py fftw.f90
```

and compile it

```
gcc -shared -I/numeric/include -I`f2py -I` -L/numeric/lib -ldfftw \
    -o fftwmodule.so -DNO_APPEND_FORTRAN fftwmodule.c wrap_dfftw.c
```

In Python:

```
>>> from Numeric import *
>>> from fftw import *
>>> print dfftw_one.__doc__
Function signature:
  out = dfftw_one(in)
Required arguments:
  in : input rank-1 array('D') with bounds (n)
Return objects:
  out : rank-1 array('D') with bounds (n)
>>> print dfftw_one([1,2,3,4])
[ 10.+0.j  -2.+2.j  -2.+0.j  -2.-2.j]
>>>
```