Interfacing Python with Fortran

Bob Dowling

University Computing Service

http://www-uxsup.csx.cam.ac.uk/courses/PythonFortran/

UCS

Outline of course

Python vs. Fortran

Fortran subroutine → Python function

Fortran 77

Numerical Python module

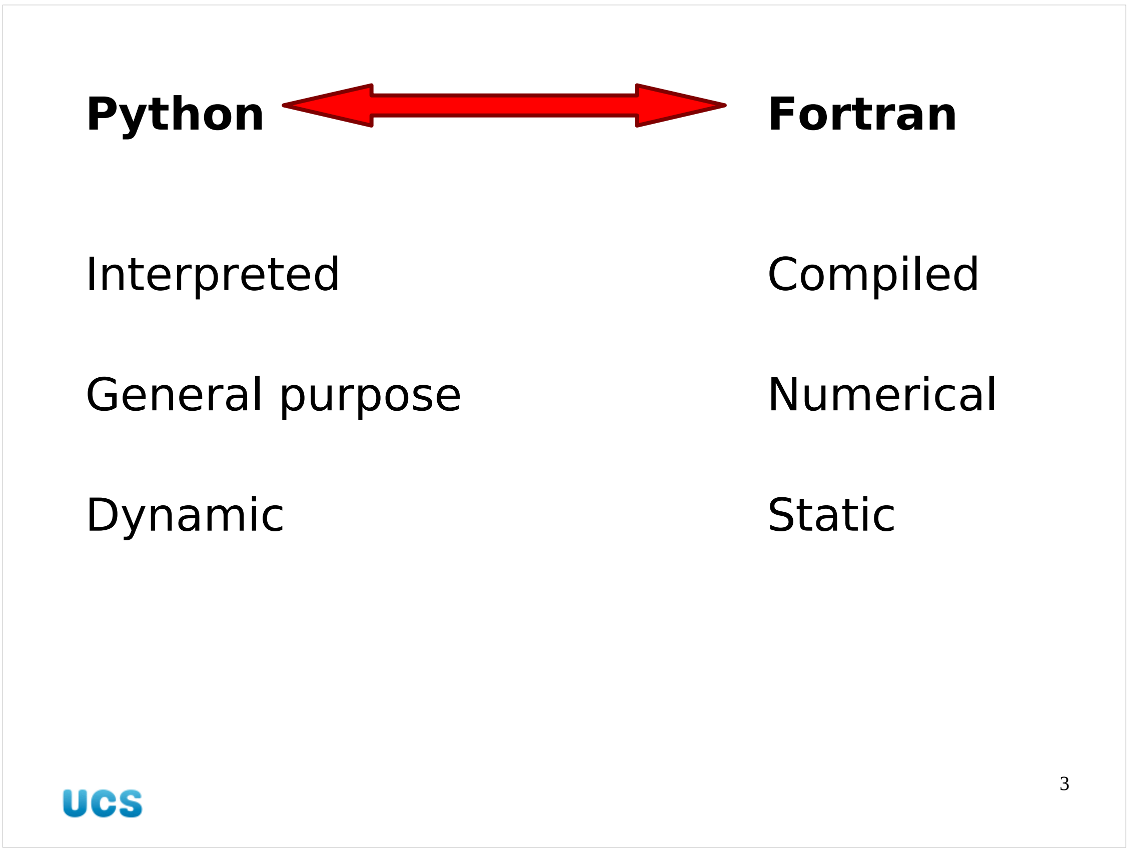
Efficiency

UCS

We start by comparing Python and Fortran, seeing them as complements rather than opposites. Then we get to work converting a Fortran subroutine into a module callable from Python.

We will work mainly in Fortran 95 but will review how to make it work with Fortran 77.

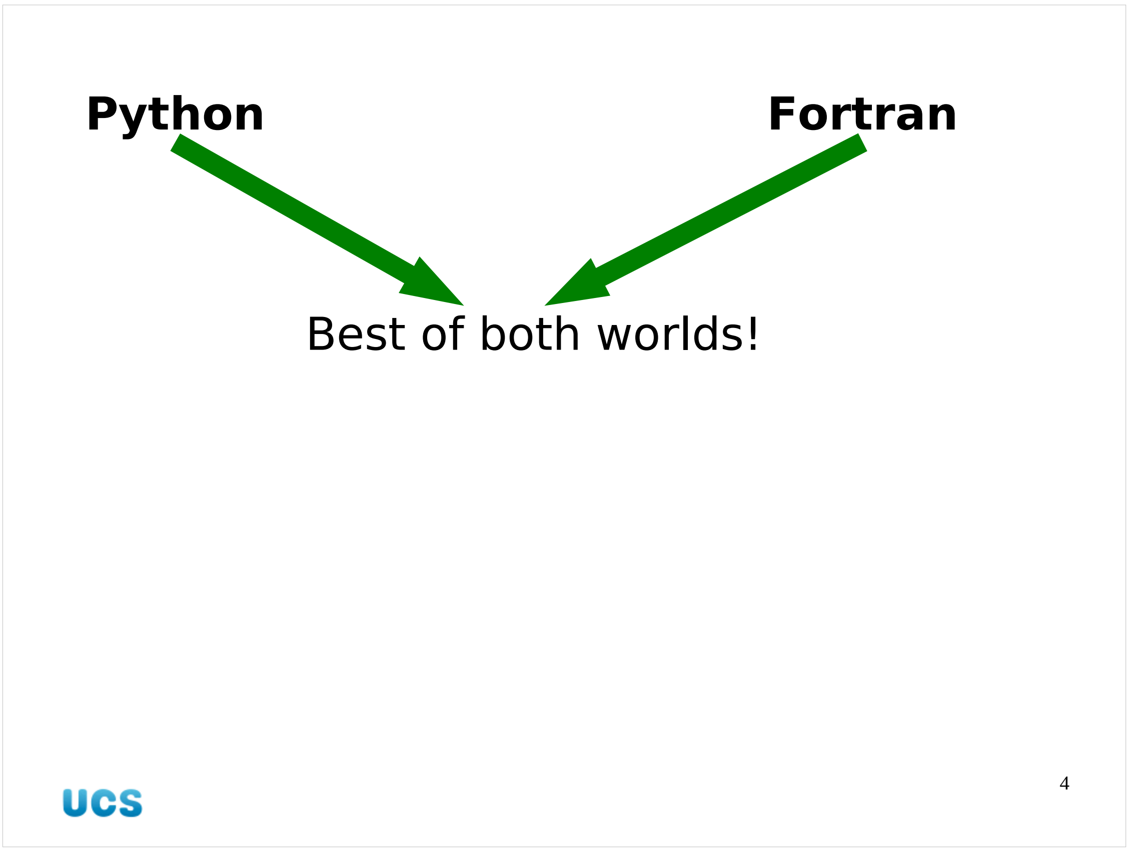
There will be a brief discussion of the underlying Numerical Python module and finally we will address a few more efficiency improvements we can make.



Python	Fortran
Interpreted	Compiled
General purpose	Numerical
Dynamic	Static

Python is a general purpose scripting language. It's excellent for gluing components of a task together and for high-level programming. It's dynamic nature (variables don't need to be declared up front; Python just makes sure they have what it needs as they go along) makes it very easy to use for quick programs. But it is an interpreted scripting language; it cannot compete with languages compiled down to machine code for speed.

Fortran is such a language. It compiles to machine code and the design of the language means it can be optimized very well. It is designed for numerical work (**For**mula **Tran**slation) and, despite the ongoing criticisms from the computing language snobs, has shown its worth by surviving and adapting over the past *fifty* years since its creation in 1957 (the first Fortran compiler). It is not, however, a general purpose programming language.

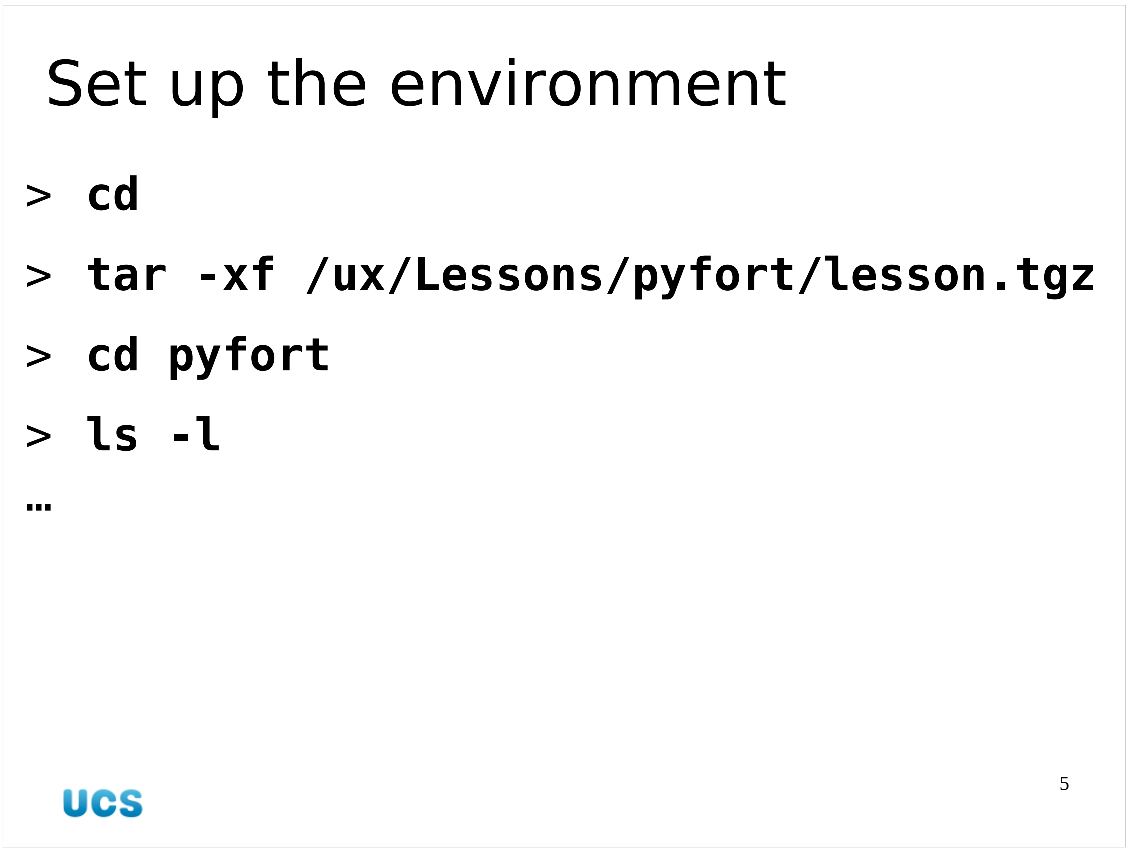


Python **Fortran**

Best of both worlds!

But there is no reason why we cannot get the best of both worlds. Python and Fortran should not be thought of as in opposition but as complements for one another.

In this course we will write our high-level program in Python and call Fortran subroutines for the numerically intensive elements.



Set up the environment

```
> cd
> tar -xf /ux/Lessons/pyfort/lesson.tgz
> cd pyfort
> ls -l
…
```

We will start by setting up a directory in our home directories ready for this course.

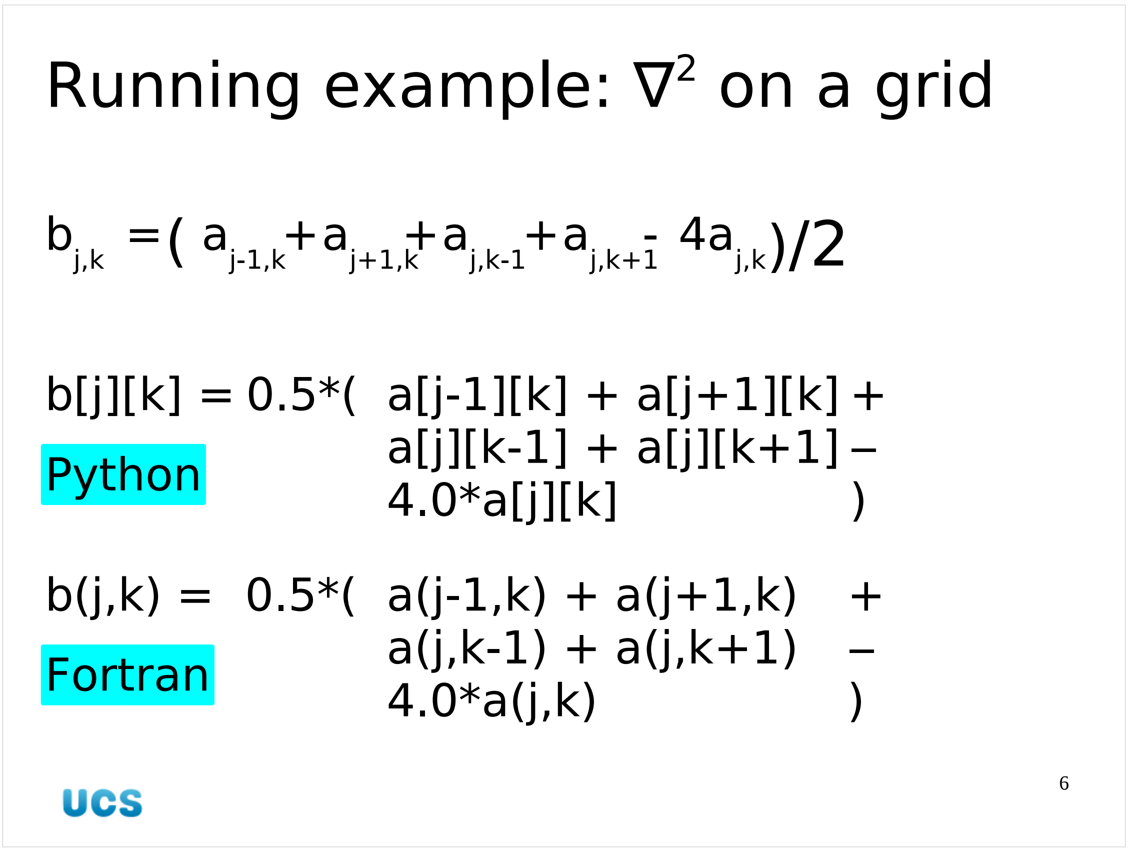
We are following the usual font conventions where

>	represents the system prompt,
>>>	represents the Python prompt,
command	bold face represents what you type, and
response	plain text represents the computer's response.

If you are following these notes in the class do this:

```
> cd
> tar -xzf /ux/Lessons/pyfort/lesson.tgz
> cd pyfort
```

If you are following them off-line then you can download the lesson file from the web at `http://www-uxsup.csx.cam.ac.uk/courses/pyfort/lesson.tgz` to a file `lesson.tgz` in your home directory and unpack that instead.



Running example: ∇^2 on a grid

$$b_{j,k} = (a_{j-1,k}+a_{j+1,k}+a_{j,k-1}+a_{j,k+1} - 4a_{j,k})/2$$

Python

```
b[j][k] = 0.5*(  a[j-1][k] + a[j+1][k] +
                 a[j][k-1] + a[j][k+1] –
                 4.0*a[j][k]            )
```

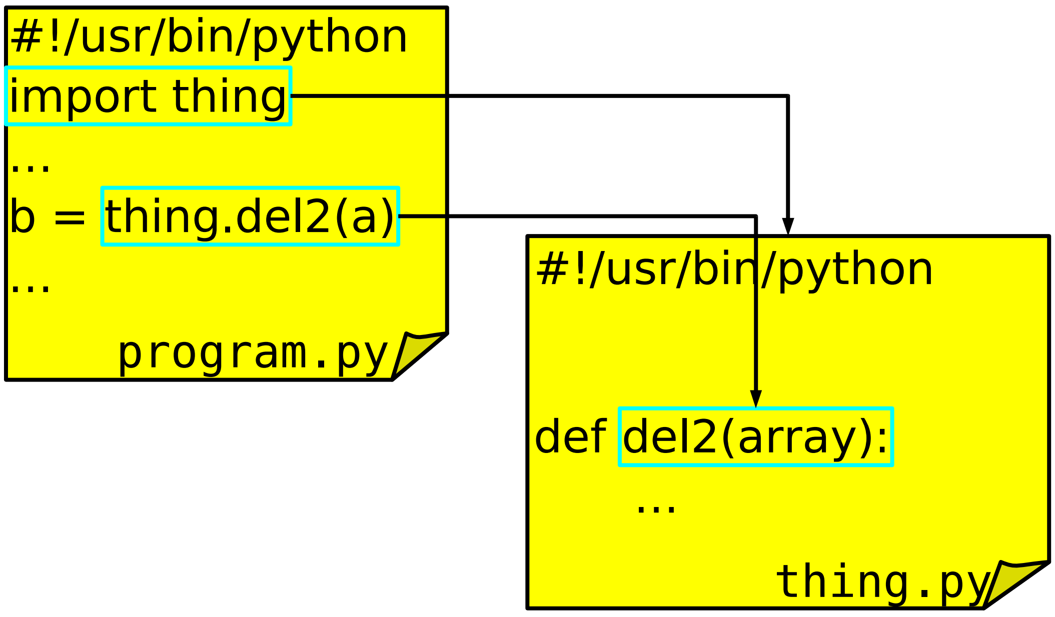
Fortran

```
b(j,k) =  0.5*(  a(j-1,k) + a(j+1,k)   +
                 a(j,k-1) + a(j,k+1)   –
                 4.0*a(j,k)            )
```

UCS

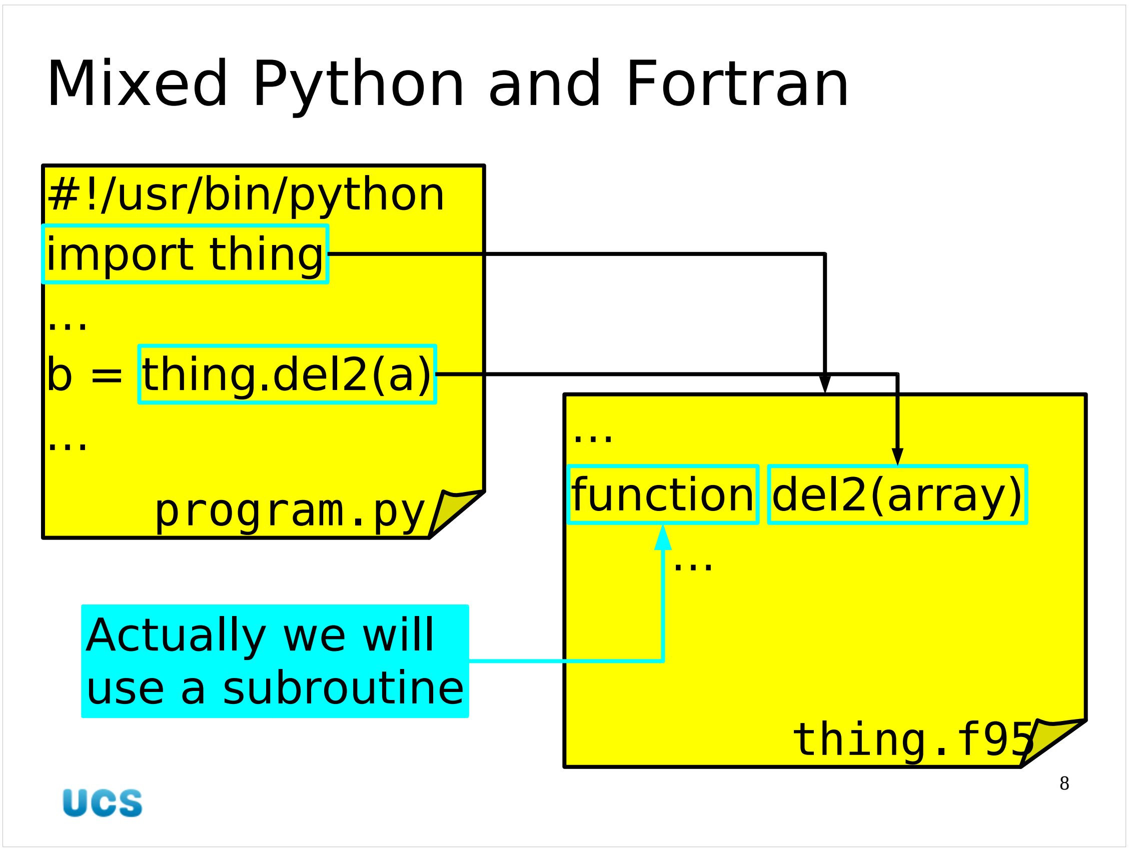
As our running example of a numerical routine we will take an easy case so that we don't distract ourselves with numerical detail. We will calculate the discrete “∇^2” on a rectangular grid. The time taken to do this scales linearly with the total size of the array being processed.

Pure Python



We could do this with pure Python. We might imagine (and soon we will see) one Python script (`program.py`, say) containing the high level logic of a program and a second one (`thing.py`, say) containing a module of the various numerical routines. The Python interpreter would run through `program.py`, calling routines out of `thing.py` as needed.

Splitting the numerical functions out to a separate file is not contrived; it makes perfect sense as you may want to use these routines in other Python programs.



Mixed Python and Fortran

```
#!/usr/bin/python
import thing
…
b = thing.del2(a)
…
```

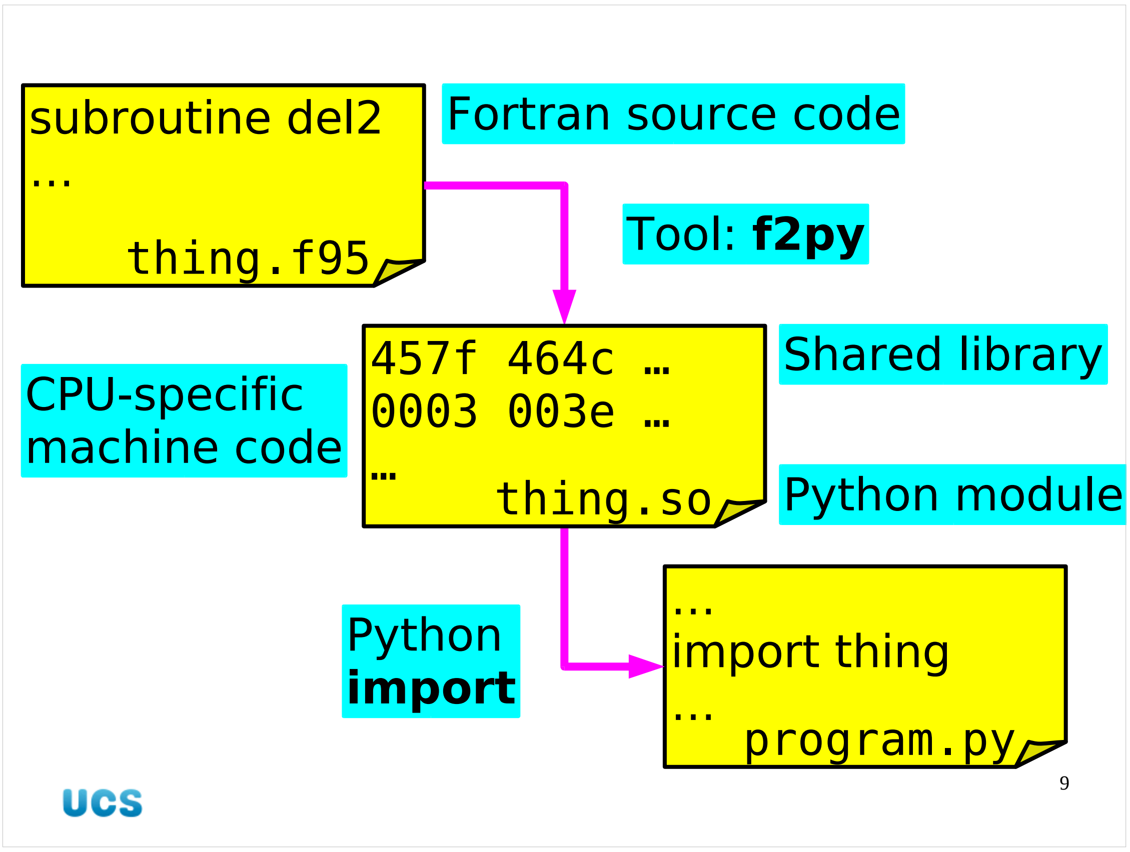
program.py

```
…
function del2(array)
  …
```

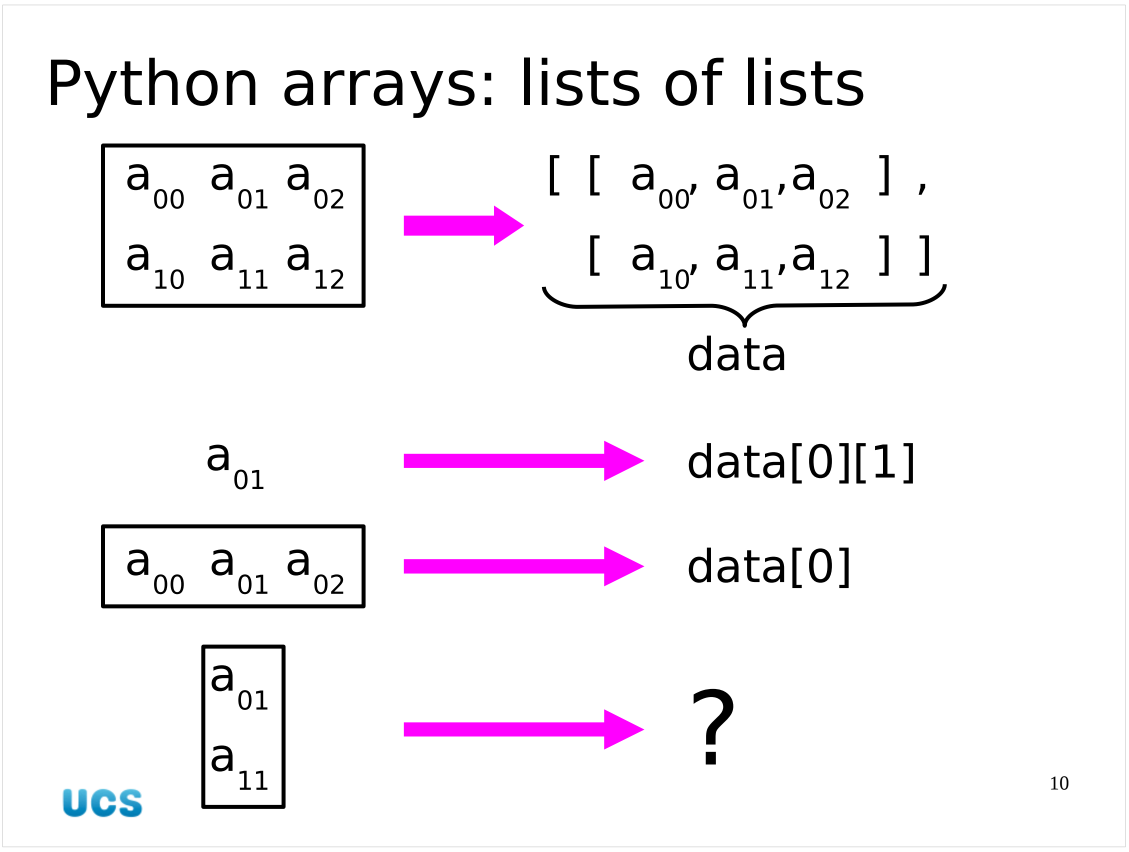
thing.f95

Our aim will be to replace the Python module with a set of Fortran files so that the numerical routines can be written in Fortran and called from Python as if it was just another module. In practice it won't be quite as simple as our ideal shown in the slide, but it won't be too bad.

(The example shows a Fortran 95 program. We can use Fortran 77 or Fortran 95 but in this course we will work in a contemporary Fortran.)



Of course Fortran is a compiled language, so Python can't interpret directly from the source code. Instead we take the Fortran file and compile it using a special program called `f2py` which creates a dynamic library (“shared object”) file which contains native machine code (which gives it the Fortran speed) but whose functions have the interfaces of a Python module (which lets us call it from Python).



Python arrays: lists of lists

$$\boxed{\begin{array}{ccc} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{array}} \rightarrow \underbrace{\begin{array}{l} [\ [\ a_{00},\ a_{01}, a_{02}\ ]\ , \\ \ \ \ [\ a_{10},\ a_{11}, a_{12}\ ]\ ] \end{array}}_{\text{data}}$$

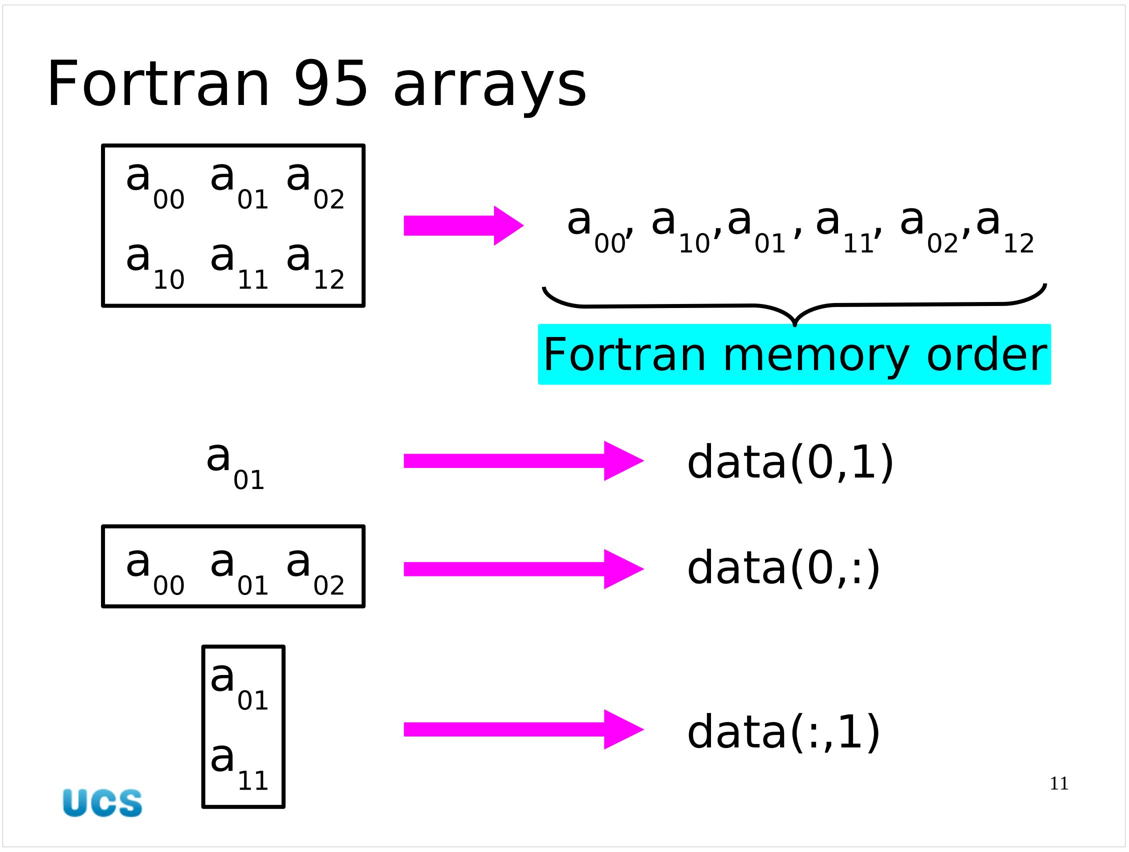
$$a_{01} \longrightarrow \text{data[0][1]}$$

$$\boxed{a_{00}\ \ a_{01}\ \ a_{02}} \longrightarrow \text{data[0]}$$

$$\boxed{\begin{array}{c} a_{01} \\ a_{11} \end{array}} \longrightarrow ?$$

We're going to be throwing about arrays of numerical data in this course, so we will start by reminding ourselves how Python normally handles arrays of data. Suppose we have a two dimensional grid of data points. Mathematically we would represent the data as a_{jk} where *j* marks the row and *k* the column. In compliance with the Python numbering convention we start counting at *0*.

Python has lists. So our array is treated in Python as a list of rows. Each of those rows is a list of the values in that row. This makes it easy to refer to a row of data at a time, but harder to refer to a column.



Fortran 95 arrays

$$\boxed{\begin{matrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{matrix}} \rightarrow \underbrace{a_{00}, a_{10}, a_{01}, a_{11}, a_{02}, a_{12}}_{\text{Fortran memory order}}$$

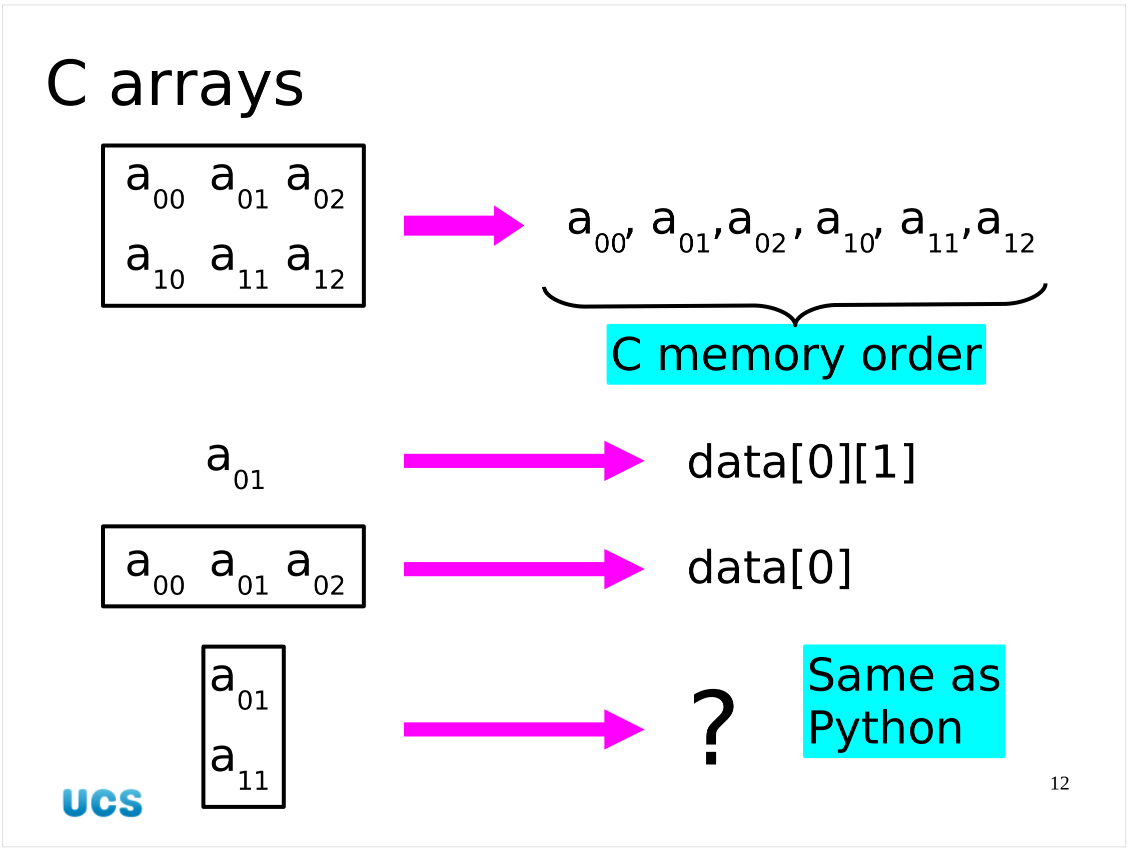
$$a_{01} \longrightarrow \text{data(0,1)}$$

$$\boxed{\begin{matrix} a_{00} & a_{01} & a_{02} \end{matrix}} \longrightarrow \text{data(0,:)}$$

$$\boxed{\begin{matrix} a_{01} \\ a_{11} \end{matrix}} \longrightarrow \text{data(:,1)}$$

Fortran has contiguous blocks of memory to store arrays. The Fortran compiler works out where in that block any particular reference is. References to rows and columns etc. are converted into start positions, end positions and “strides” – the number of elements in memory to skip each time. Fortran can refer to rows and columns with equal ease, therefore.

Note the order in which the array data is laid out in memory. The first index varies fastest (i.e. every item).



C arrays

a_{00} a_{01} a_{02}
a_{10} a_{11} a_{12}
$a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{12}$
C memory order
a_{01}
data[0][1]
a_{00} a_{01} a_{02}
data[0]
a_{01}
a_{11}
?
Same as Python

C stores arrays in a way very similar to Python's list of lists. Each list is laid down in memory one after the other. A row is identified as an offset into this block of elements. There is no way to refer to a column. In this it has exactly the same limitations as Python.

Note that the order of elements in a C array is different from the Fortran order. In C the first index varies slowest.

Python trick for creating arrays

```
>>> data = [ x**2 for x in range(0, 5) ]
>>> data
[0, 1, 4, 9, 16]


>>> data = [ 0 for x in range(0, 5) ]
>>> data
[0, 0, 0, 0, 0]
```

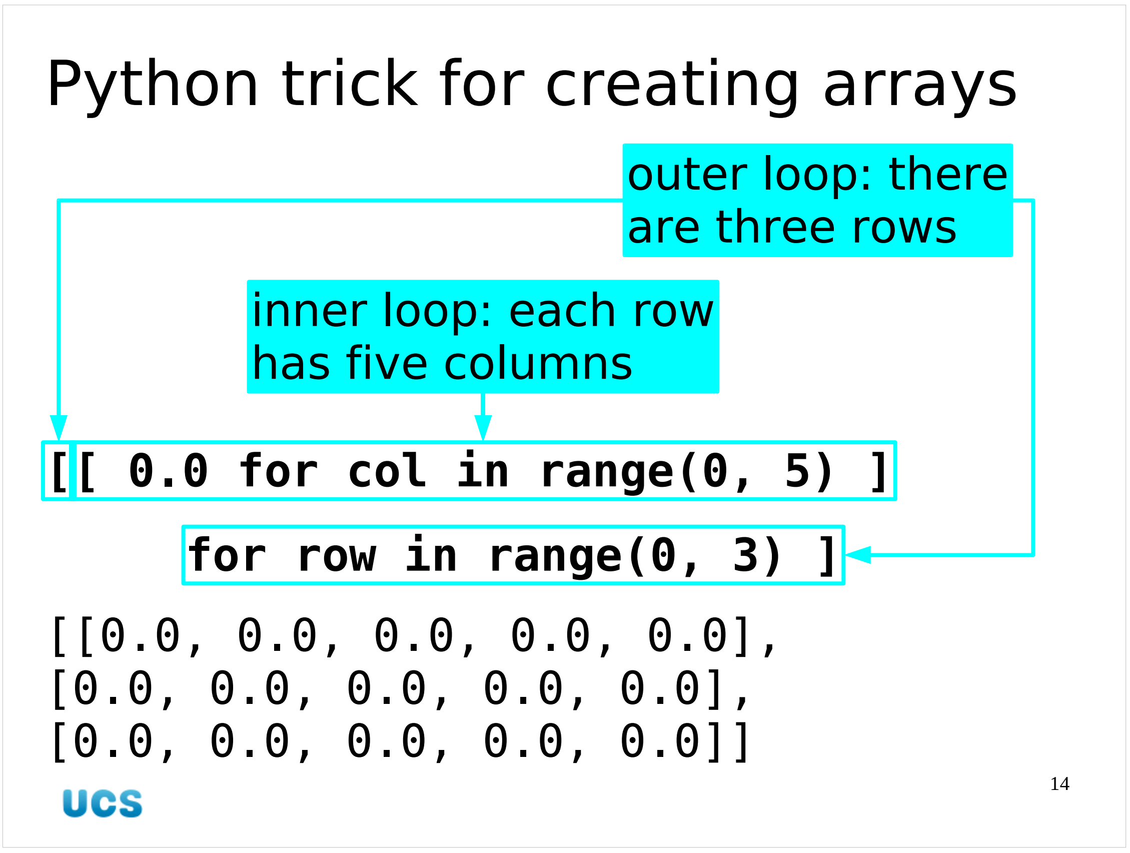
Given that we are going to be working with arrays we should see a simple Python trick for creating arrays simply. Python has a neat syntax for creating lists of values by twisting the usual `for` loop inside a list.

It was designed to evaluate functions for values in lists:

```
>>> [ x**2 for x in range(0,10) ]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [ 'm%sss' % x for x in ['a', 'e', 'i', 'o', 'u'] ]
['mass', 'mess', 'miss', 'moss', 'muss']
```

but there is no reason why it can't be used with constant values to initialise a list of zeroes, say:

```
>>> [ 0.0 for x in range(0,10) ]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```



Python trick for creating arrays

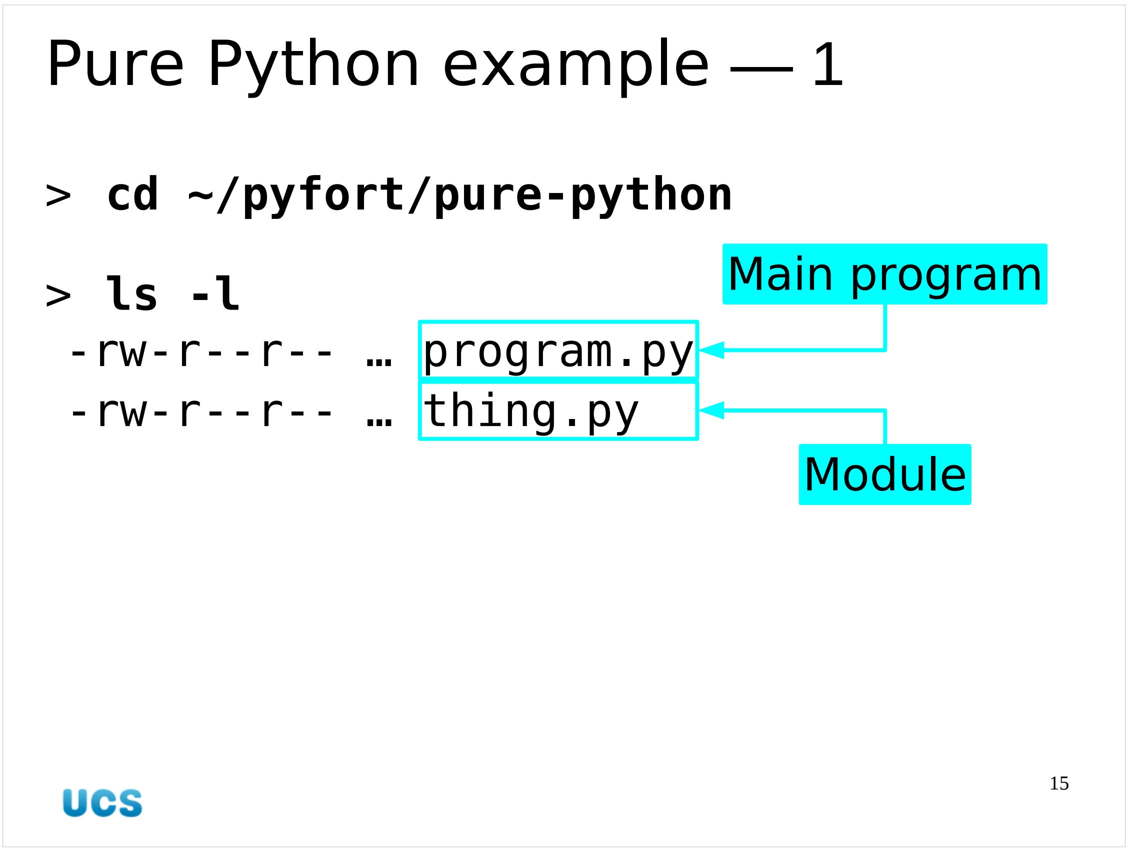
outer loop: there are three rows
inner loop: each row has five columns

```
[[ 0.0 for col in range(0, 5) ]
      for row in range(0, 3) ]
```

```
[[0.0, 0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0]]
```

That list can be regarded as one row so we can repeat the trick to create a list of those lists:

```
>>> [ [ 0.0 for x in range(0,10) ] for y in range(0,5) ]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0]]
```

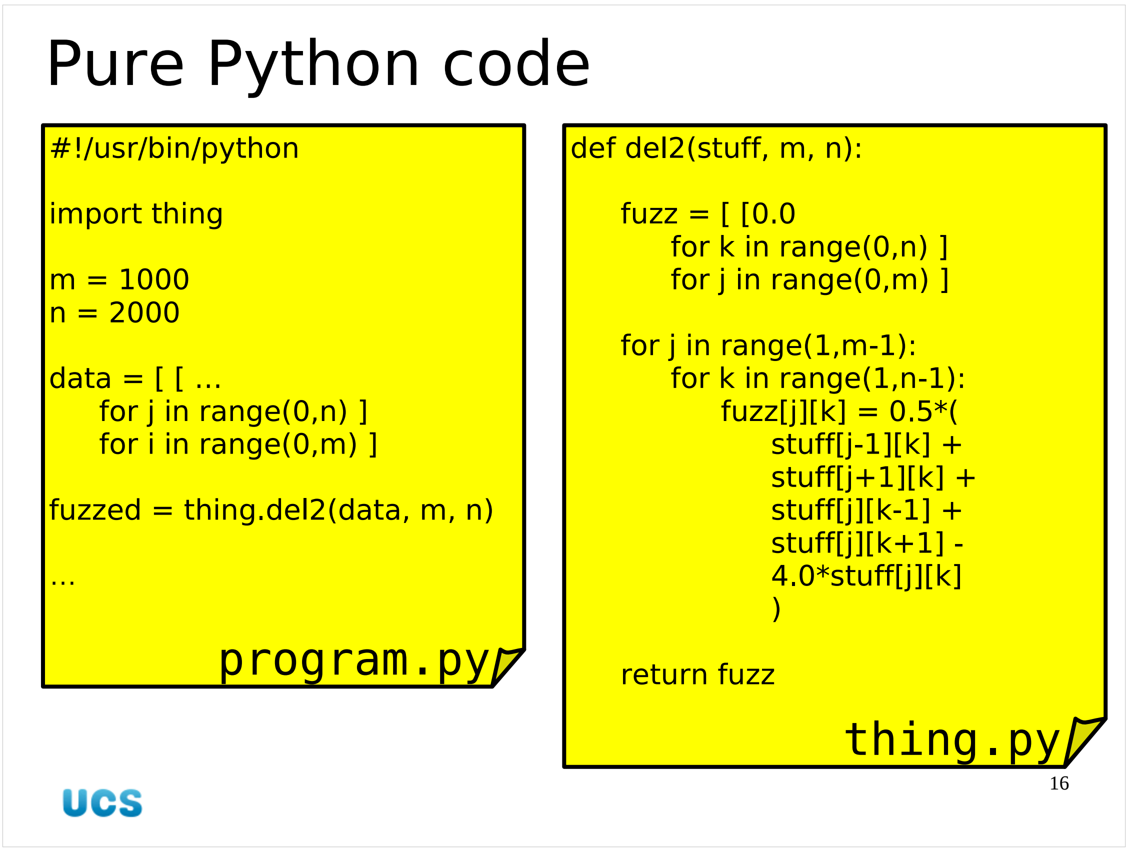


Pure Python example — 1

```
> cd ~/pyfort/pure-python

> ls -l
 -rw-r--r-- … program.py
 -rw-r--r-- … thing.py
```

Main program
Module
UCS

So let's get started with a pure Python example. In the directory `~/pyfort/pure-python` is a simple program that spits out some results which don't interest us. What matters is that it has the two-way split of a program file and a module file.



Pure Python code

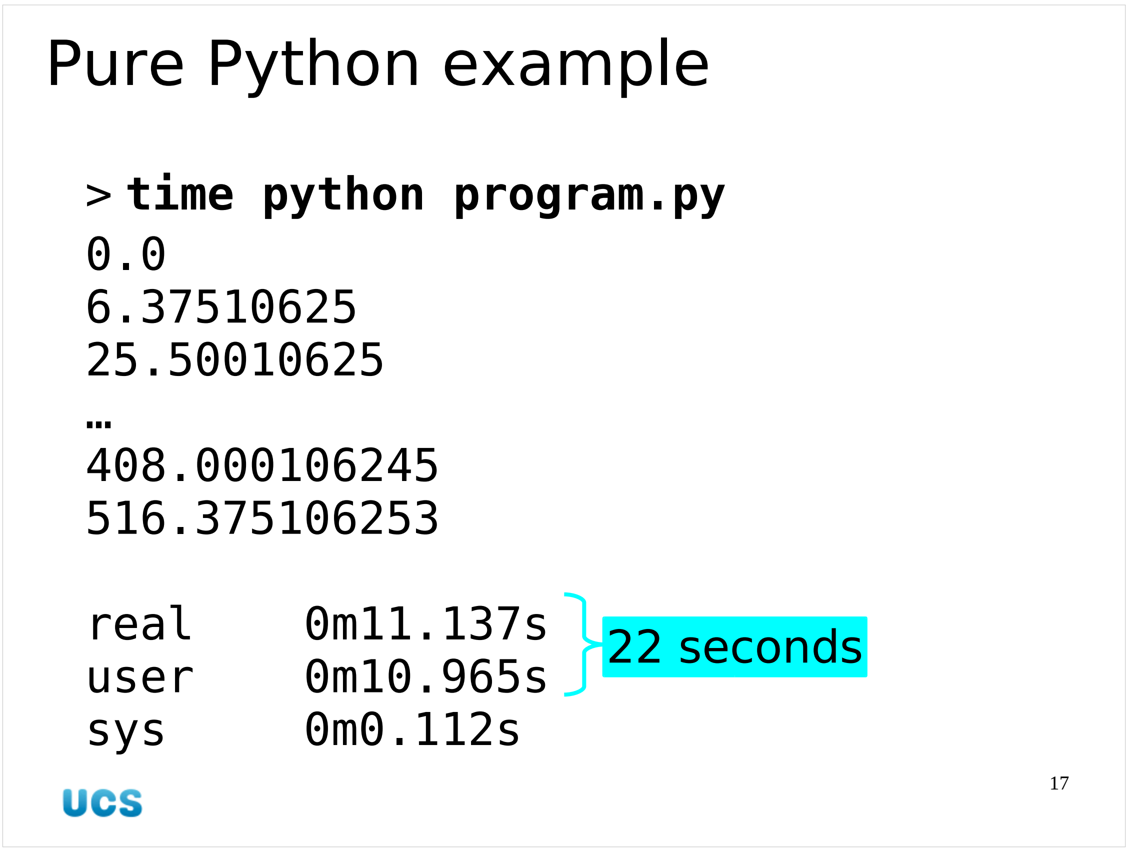
```
#!/usr/bin/python

import thing

m = 1000
n = 2000

data = [ [ …
     for j in range(0,n) ]
     for i in range(0,m) ]

fuzzed = thing.del2(data, m, n)

…
```

program.py

```
def del2(stuff, m, n):

     fuzz = [ [0.0
          for k in range(0,n) ]
          for j in range(0,m) ]

     for j in range(1,m-1):
          for k in range(1,n-1):
               fuzz[j][k] = 0.5*(
                    stuff[j-1][k] +
                    stuff[j+1][k] +
                    stuff[j][k-1] +
                    stuff[j][k+1] -
                    4.0*stuff[j][k]
                    )

     return fuzz
```

thing.py

The code involved is not that complex for our simple example. The module defines the function that calculates ∇^2 and the program calls it.



Pure Python example

```
> time python program.py
0.0
6.37510625
25.50010625
…
408.000106245
516.375106253

real    0m11.137s
user    0m10.965s
sys     0m0.112s
```

22 seconds

We can run the command and also time it.

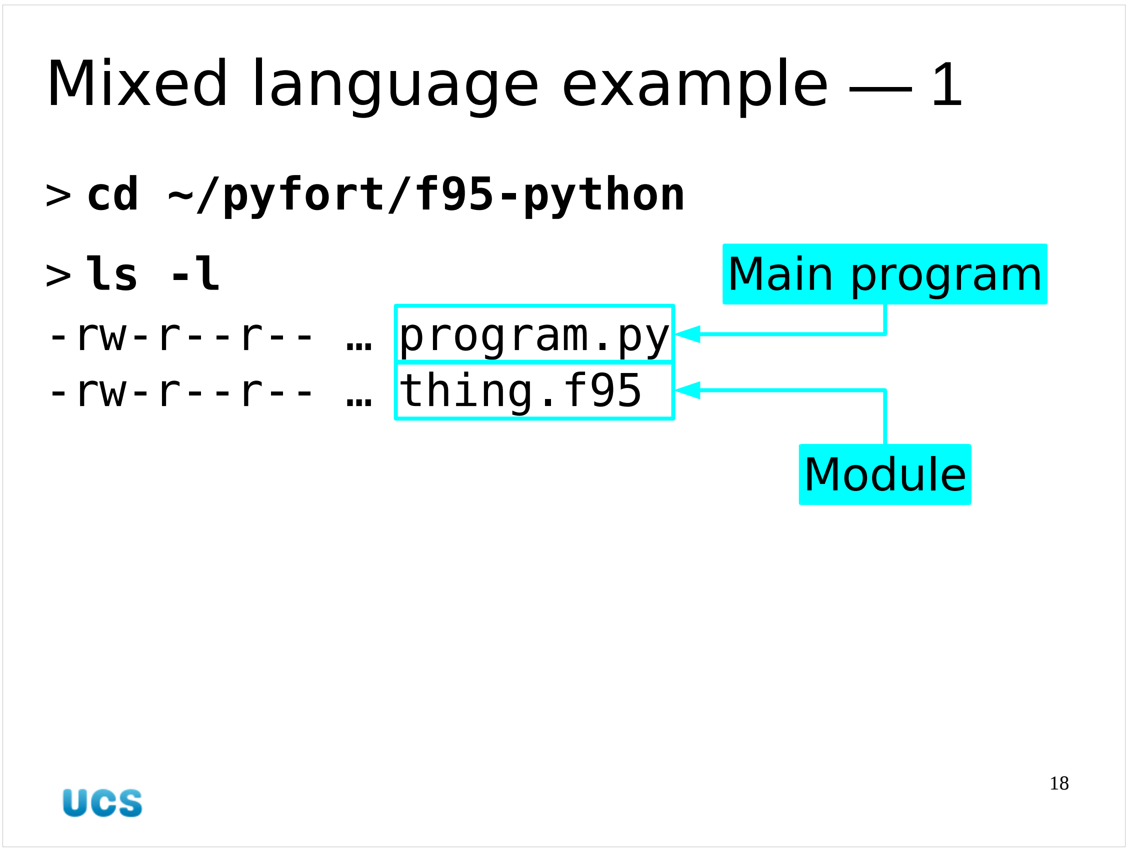
When we come to work on the mixed Python/Fortran example, we will compare the results we get with those generated here. We will also compare the time taken.

For those of you who have not seen the “`time`” command before, it yields three measures of how long a command took to run.

“Real time” is the elapsed, clock on the wall time taken. This includes the time when your command wasn't running because the computer was running somebody else's program. It's just the difference between end and start times.

“User time” is the measure of how much system time went into running your code.

“System time” is the amount of system time that was taken up by the core operating system (the kernel) on behalf of your code.



Mixed language example — 1

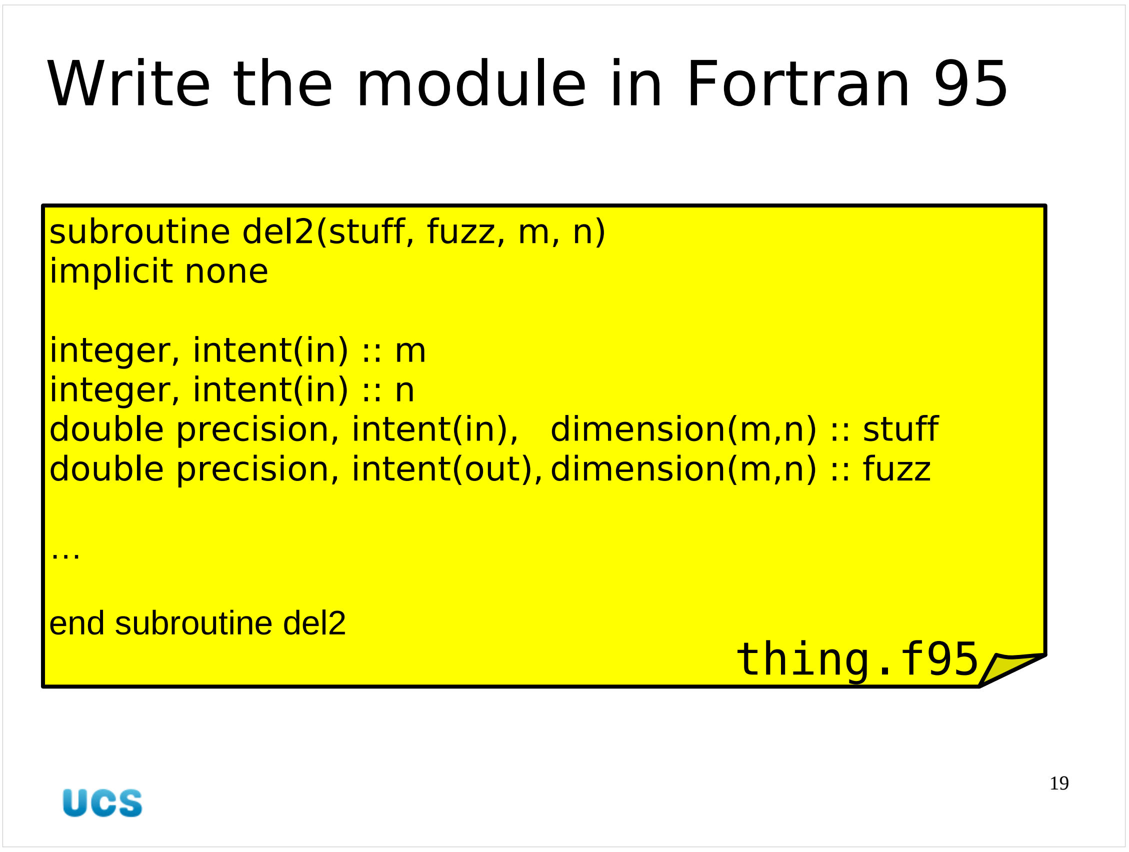
```
> cd ~/pyfort/f95-python
> ls -l
-rw-r--r-- … program.py
-rw-r--r-- … thing.f95
```

Main program

Module

UCS

Now let's look at an example of mixing Python and Fortran files. The directory `~/pyfort/f95-python` contains two files. The `program.py` file is exactly the same as the file in the `pure-python` directory. But instead of a `thing.py` Python file there is a `thing.f95` Fortran source code file.



Write the module in Fortran 95

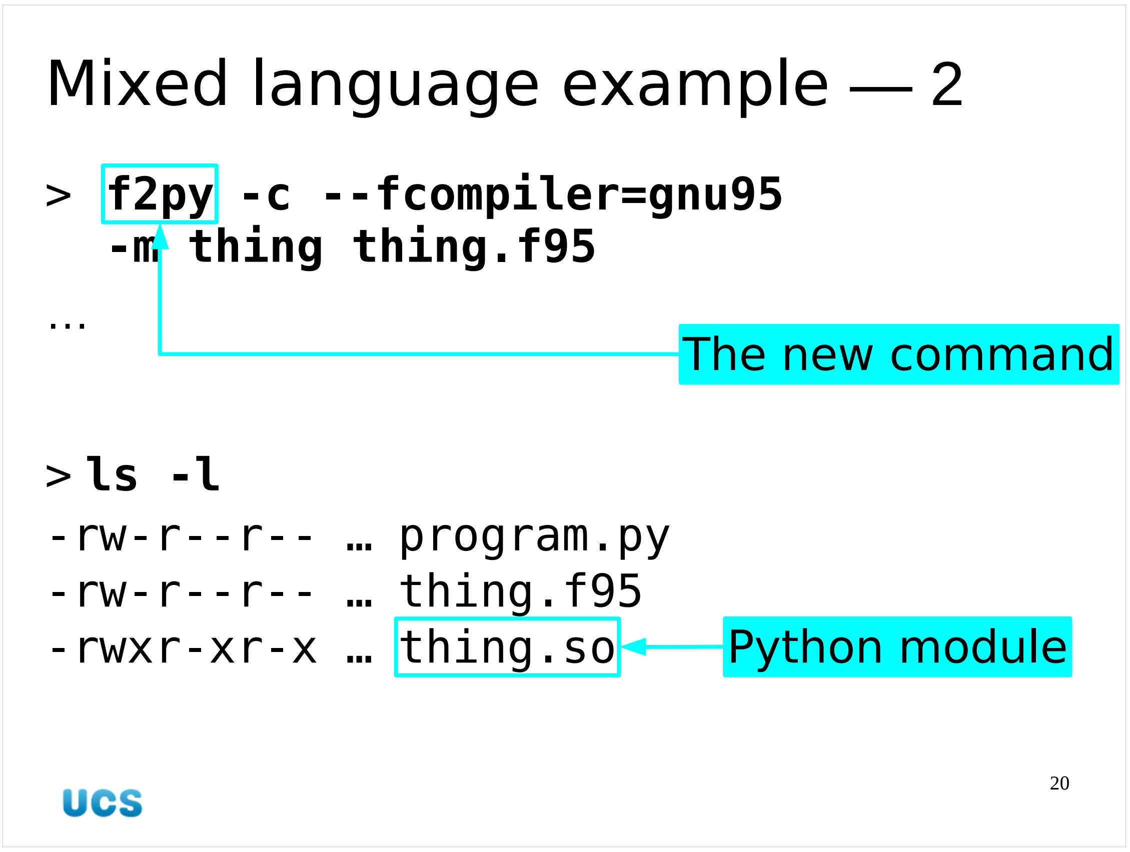
```
subroutine del2(stuff, fuzz, m, n)
implicit none

integer, intent(in) :: m
integer, intent(in) :: n
double precision, intent(in),   dimension(m,n) :: stuff
double precision, intent(out), dimension(m,n) :: fuzz

…

end subroutine del2
```

thing.f95

The `thing.f95` file contains a subroutine with the same name as the function in the `thing.py` file we saw before, `del2()`. It does not have exactly the same arguments as the Python function and is a subroutine rather than a function.

Note that we have specified the properties of the four arguments as tightly as we can in their declarations. In particular, we have specified which parameters are to be read from and which to be written into. This will matter when we come to represent this Fortran subroutine as a Python function.

Also note that we do not create the `fuzz` array. In the Fortran we assume that it has already been created and is being passed into the subroutine.



Mixed language example — 2

```
> f2py -c --fcompiler=gnu95
   -m thing thing.f95
…
```

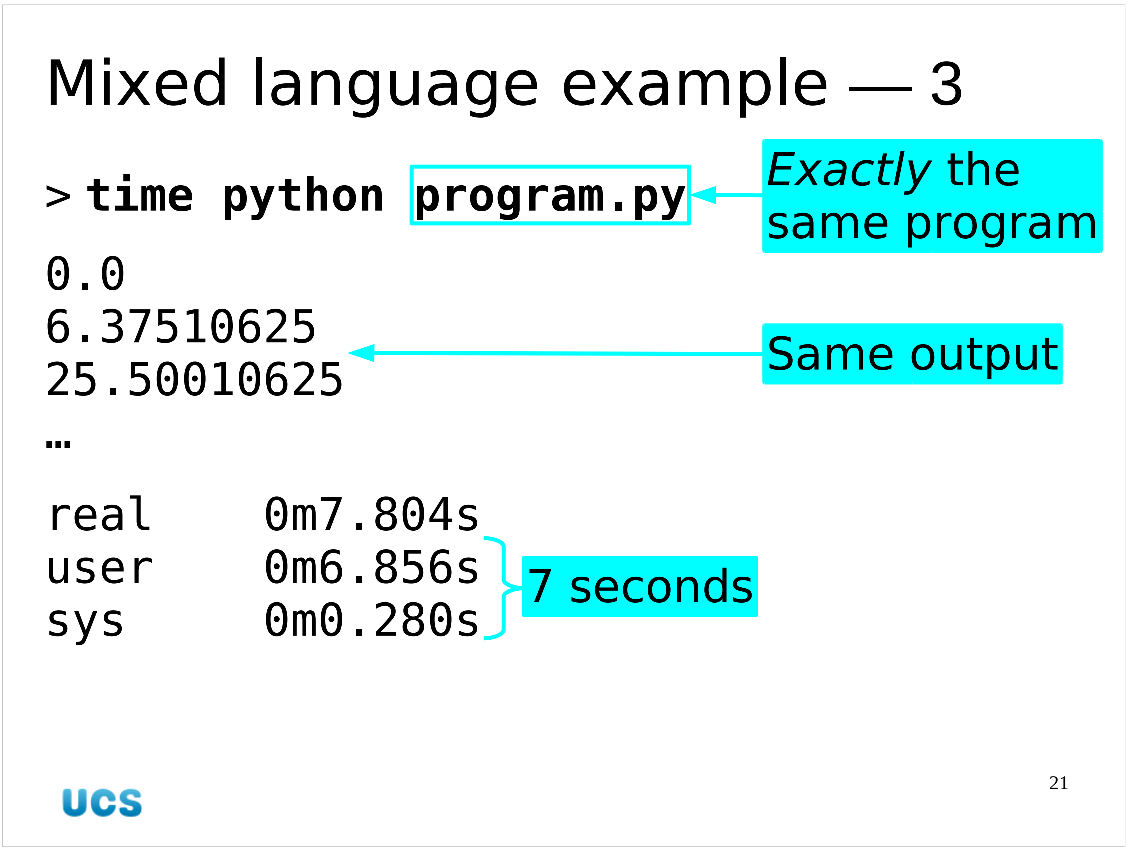
The new command

```
> ls -l
-rw-r--r-- … program.py
-rw-r--r-- … thing.f95
-rwxr-xr-x … thing.so
```

Python module

The first thing we must do is to compile the Fortran source code to machine code. We do not use the Fortran compiler directly. Instead we use a program called “`f2py`”. It requires a stack of options to do what we want it to do and we will examine those very soon. For the time being we will issue the command shown in the slide and see that it creates one additional file, `thing.so`. In machine code library terms this is a “**s**hared **o**bject”, also known as a “dynamic library”. It is a block of machine code defining some functions that multiple programs can use.

It does not, however, simply contain the Fortran subroutine `del2()` so that it could be called from another Fortran program. Instead, it contains a number of functions which correspond to those needed by a Python program. This file is a machine code representation of a Python module called “`thing`” which can be imported into a Python program.



Mixed language example — 3

```
> time python program.py
0.0
6.37510625
25.50010625
…

real    0m7.804s
user    0m6.856s
sys     0m0.280s
```

Exactly the same program

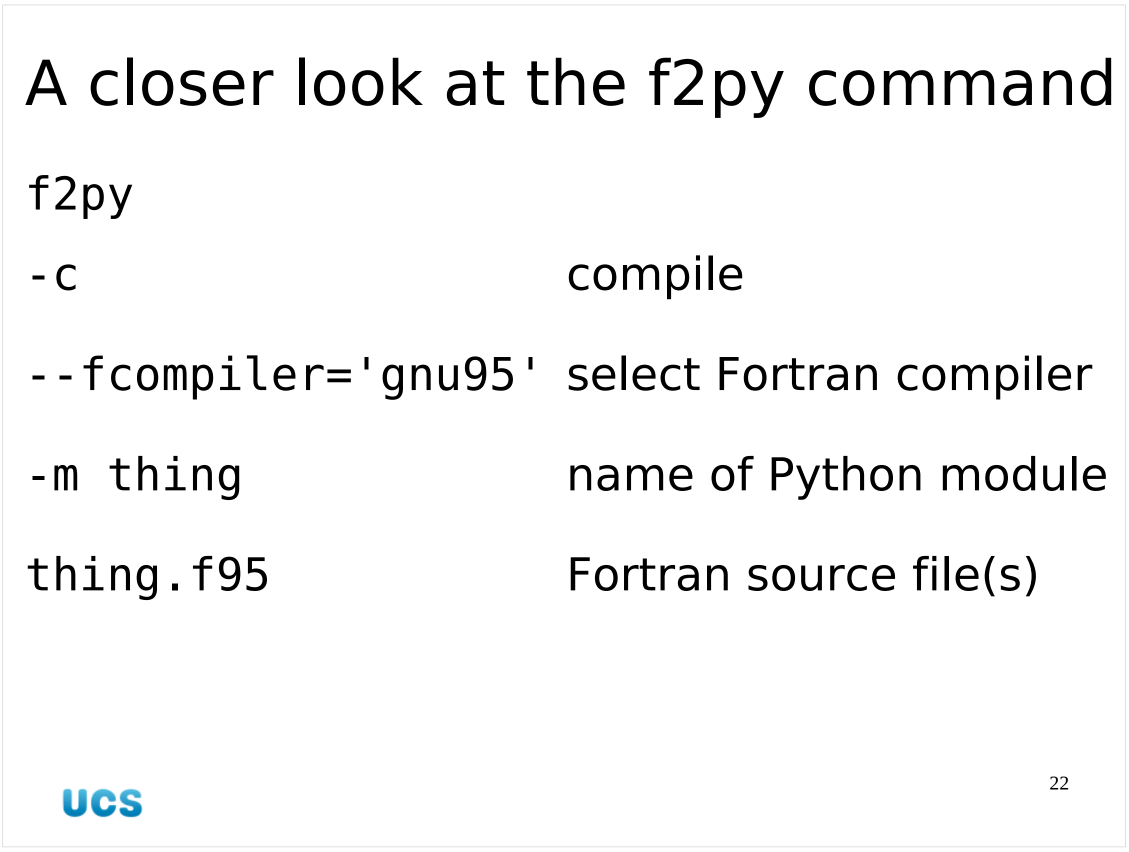
Same output

7 seconds

UCS

So now we can run our program again but this time we use a machine code module rather than a Python module.

It gives exactly the same results and is faster. (To be honest, these timing tests are way too short to give seriously meaningful data, but they are sufficient to make the point.)



A closer look at the f2py command

`f2py`

`-c`	compile
`--fcompiler='gnu95'`	select Fortran compiler
`-m thing`	name of Python module
`thing.f95`	Fortran source file(s)

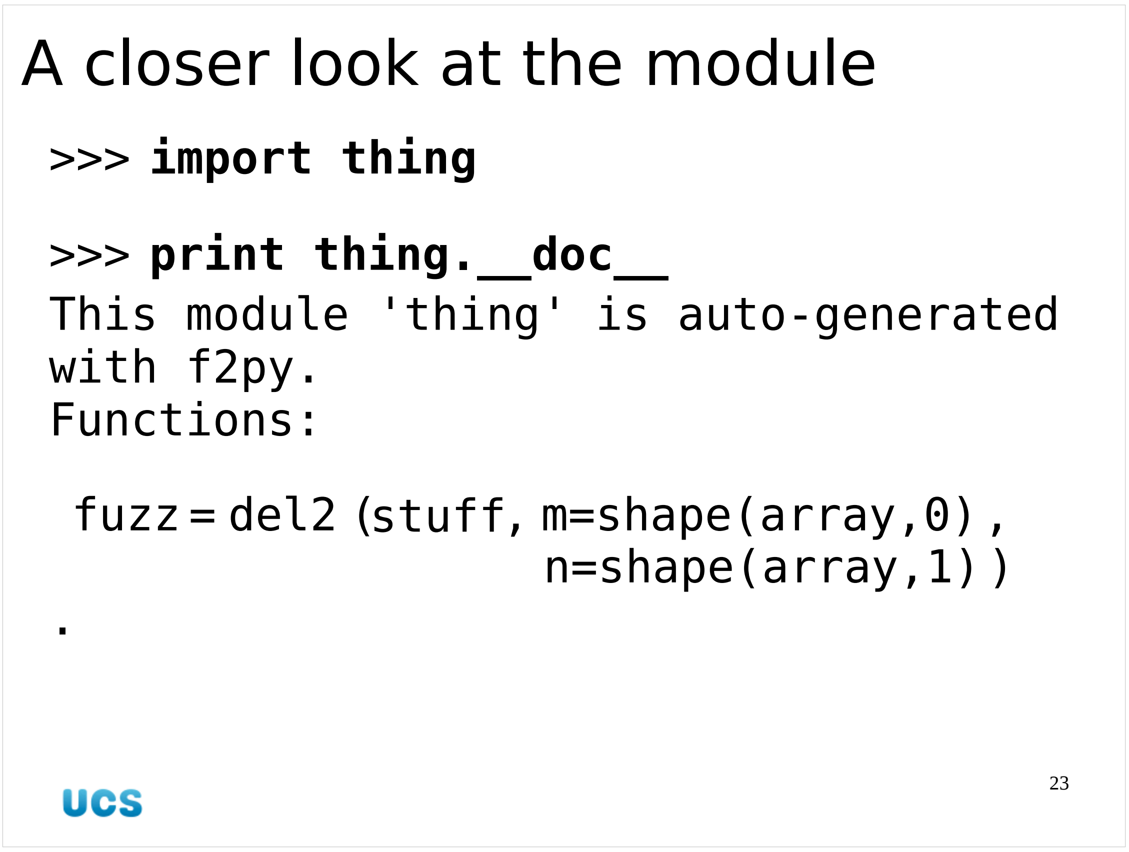
`f2py` is the name of the program, meaning “**F**ortran **to Py**thon”.

It is capable of more than just creation of the module so we explicitly tell it to compile with the “`-c`” option.

The exact options and settings required depend on the Fortran compiler being used. The option “`--fcompiler`” allows us to specify exactly which compiler and set of options to use. This does mean that `f2py` can only be used with compilers it supports but the list is long. See the appendix at the end of the notes for a full list (as of version 2.4422 of `f2py`). Alternatively, the command “`f2py -c --help-fcompiler`” will list all the compiler keys.

The “`-m`” option specifies the name of the module being created. It does not need to match the name of the Fortran source code file.

Finally we list the Fortran source code files we need to compile. In this case there is only one but it is possible to quote multiple Fortran files on the command line.

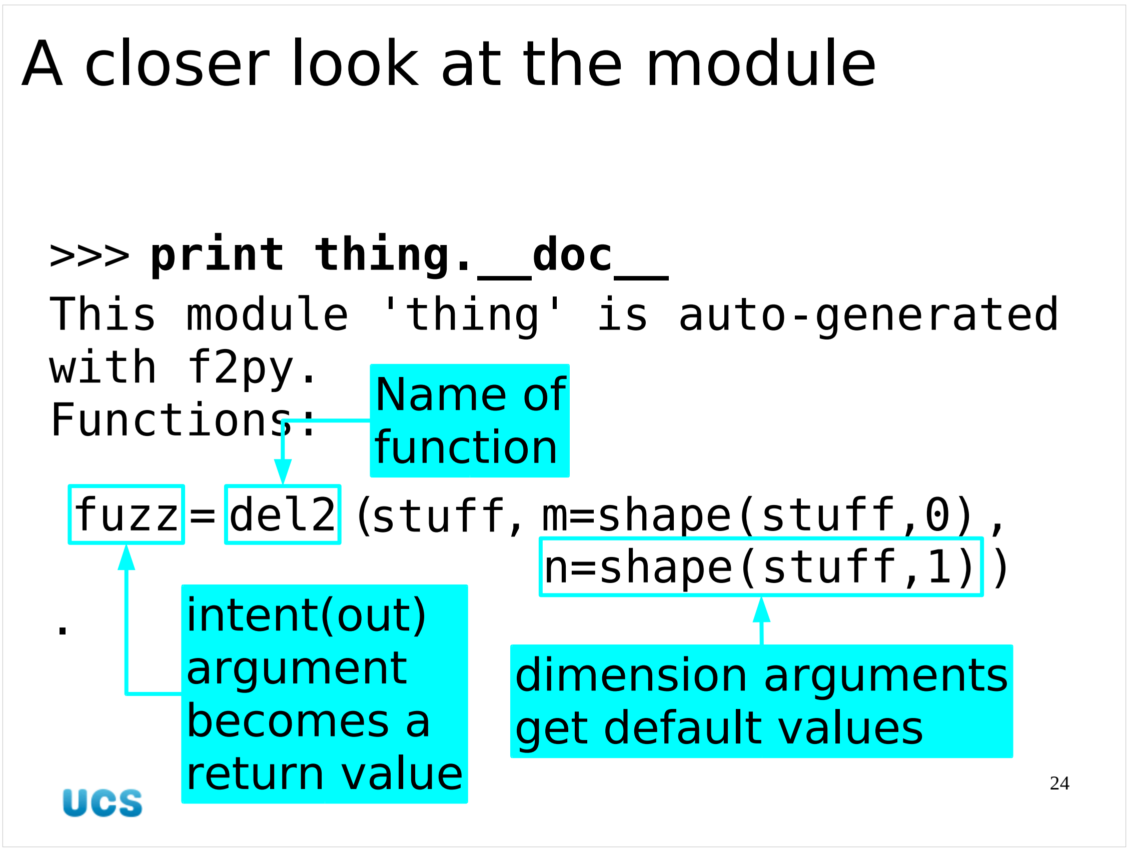


A closer look at the module

```
>>> import thing

>>> print thing.__doc__
This module 'thing' is auto-generated
with f2py.
Functions:

 fuzz = del2 (stuff, m=shape(array,0),
                     n=shape(array,1))
.
```

Now let's look at the module itself. We will launch Python interactively and import the module by hand.

The module's `__doc__` string contains the interface definitions for the functions within the module, in our case just `del2()`. There are many things to observe in this definition and we will address them now.



A closer look at the module

```
>>> print thing.__doc__
This module 'thing' is auto-generated
with f2py.
Functions:
 fuzz = del2 (stuff, m=shape(stuff,0) ,
                     n=shape(stuff,1) )
.
```

Name of function
intent(out) argument becomes a return value
dimension arguments get default values

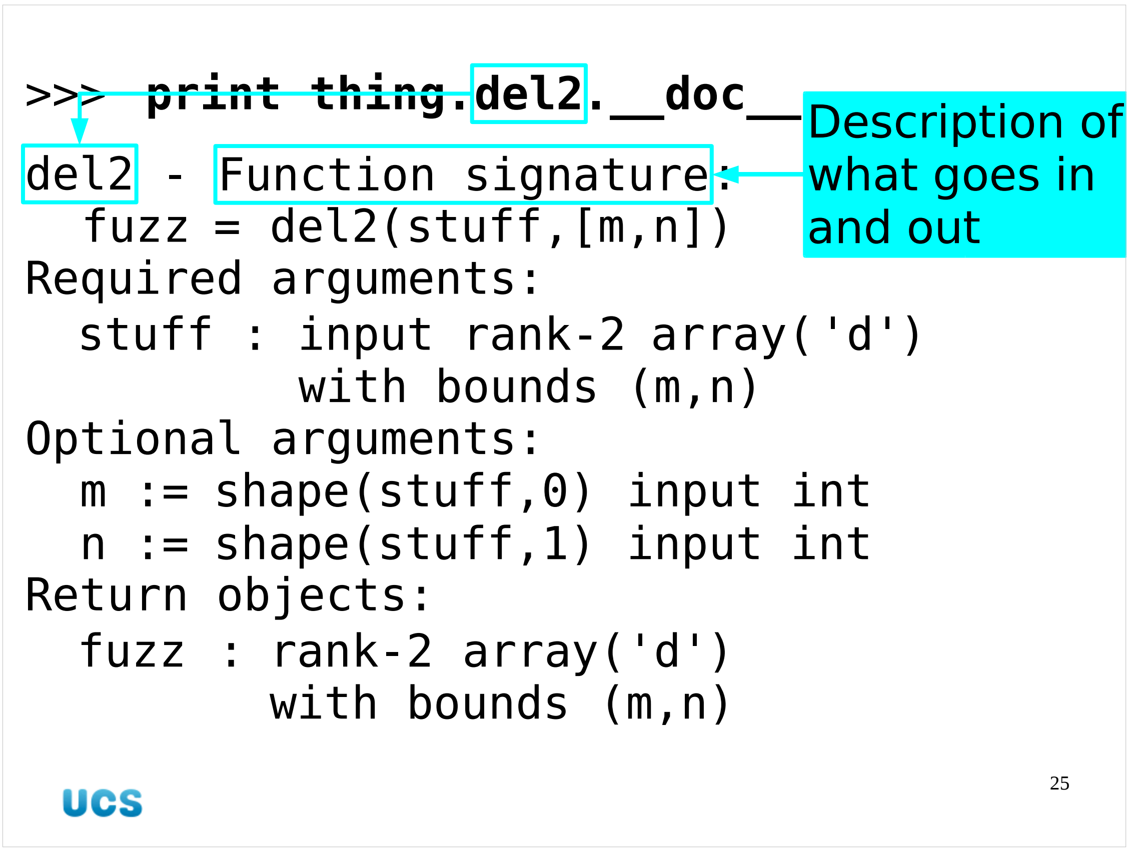
24

The first thing we notice is that we have a module function with the same name as the Fortran subroutine. Big deal.

More interestingly, we note that while the subroutine had a parameter `fuzz` which was declared with `intent(out)` the Python function has that parameter as its return value. If multiple parameters had been declared with `intent(out)` then a tuple of them would have been the function's return value.

Better still, the two dimension parameters now have (the correct) default values established.

We will see the “`shape()`” function a bit later but in a nut-shell, the shape of an array along its 0th axis is the number of rows it has and its shape along its 1st axis is its number of columns.

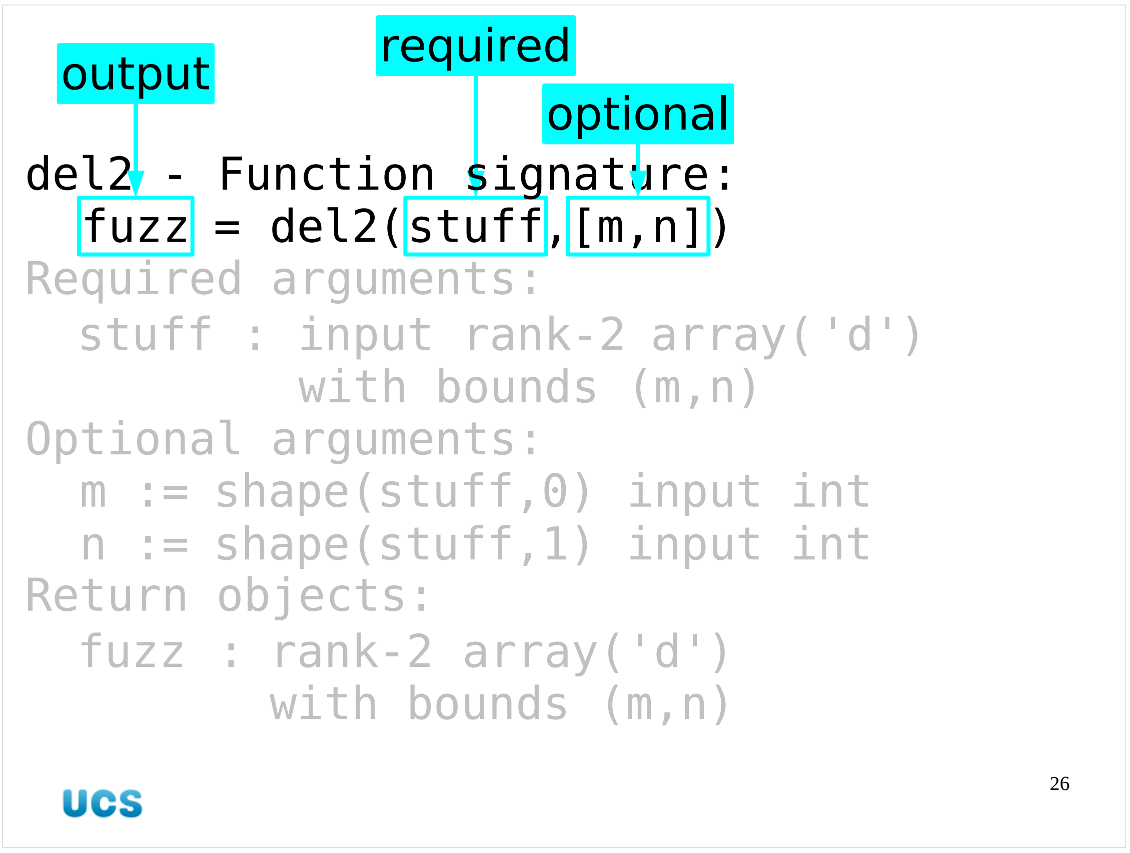


```
>>> print thing.del2.__doc__
del2 - Function signature:
  fuzz = del2(stuff,[m,n])
Required arguments:
  stuff : input rank-2 array('d')
          with bounds (m,n)
Optional arguments:
  m := shape(stuff,0) input int
  n := shape(stuff,1) input int
Return objects:
  fuzz : rank-2 array('d')
         with bounds (m,n)
```

As well as getting a summary of the functions in the module from the module's __doc__ strings, we can get detail from the function's.

The “signature” is a posh way to describe everything about the function that you need to know from the outside, while telling you nothing about what it actually does on the inside. So the signature describes what all the arguments to the function are and what type they have to be. It also describes the types of the output without describing what they will contain.

We will go through the signature created for our new `del2()` function one chunk at a time.

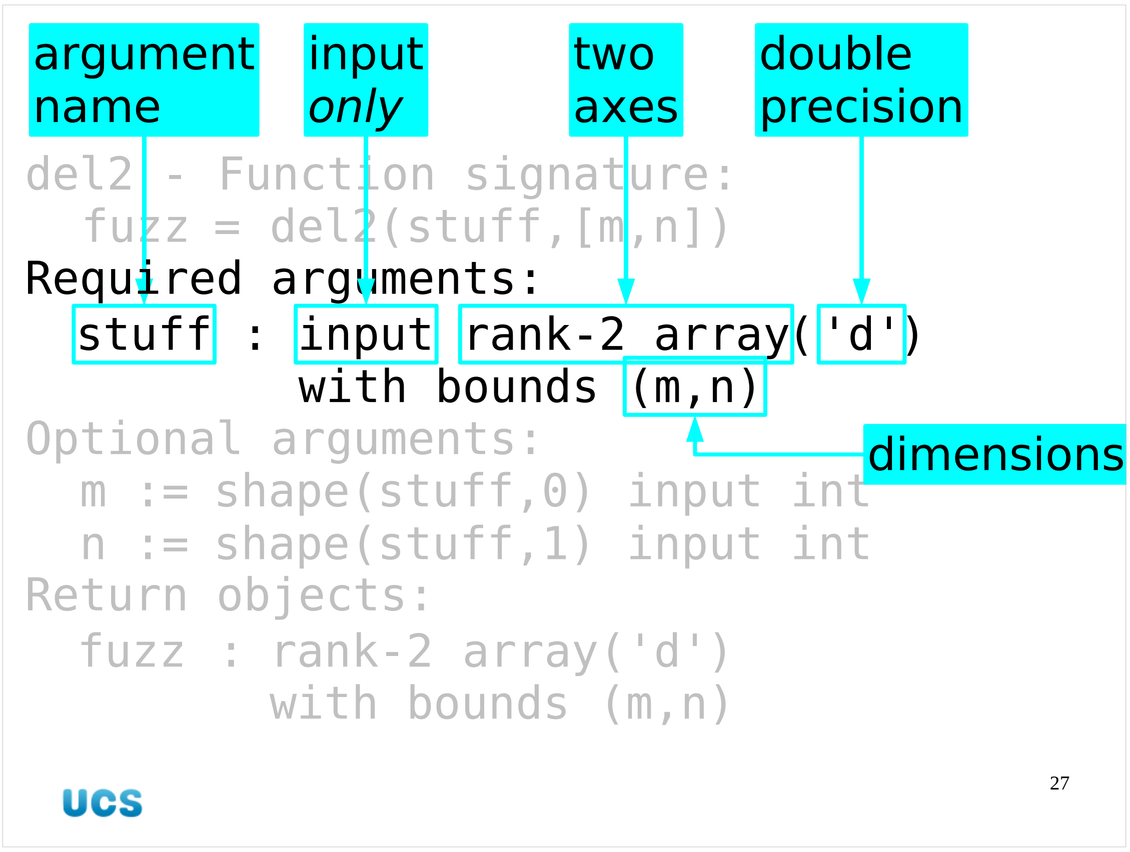

output
required
optional

```
del2 - Function signature:
  fuzz = del2(stuff,[m,n])
Required arguments:
  stuff : input rank-2 array('d')
          with bounds (m,n)
Optional arguments:
  m := shape(stuff,0) input int
  n := shape(stuff,1) input int
Return objects:
  fuzz : rank-2 array('d')
         with bounds (m,n)
```

The first line is a slightly marked up example of how the function is used from Python.

We see that the function produces a single output and takes one mandatory input. The square brackets indicate optional arguments. We still don't know what the arguments (or returned value) are yet and that's what comes next.

The arguments are given names for consistency throughout the signature definition. These name are taken from our function definition but of course there's no constraint on what names you choose to use.


argument name
input *only*
two axes
double precision
dimensions

```
del2 - Function signature:
  fuzz = del2(stuff,[m,n])
Required arguments:
  stuff : input rank-2 array('d')
          with bounds (m,n)
Optional arguments:
  m := shape(stuff,0) input int
  n := shape(stuff,1) input int
Return objects:
  fuzz : rank-2 array('d')
         with bounds (m,n)
```

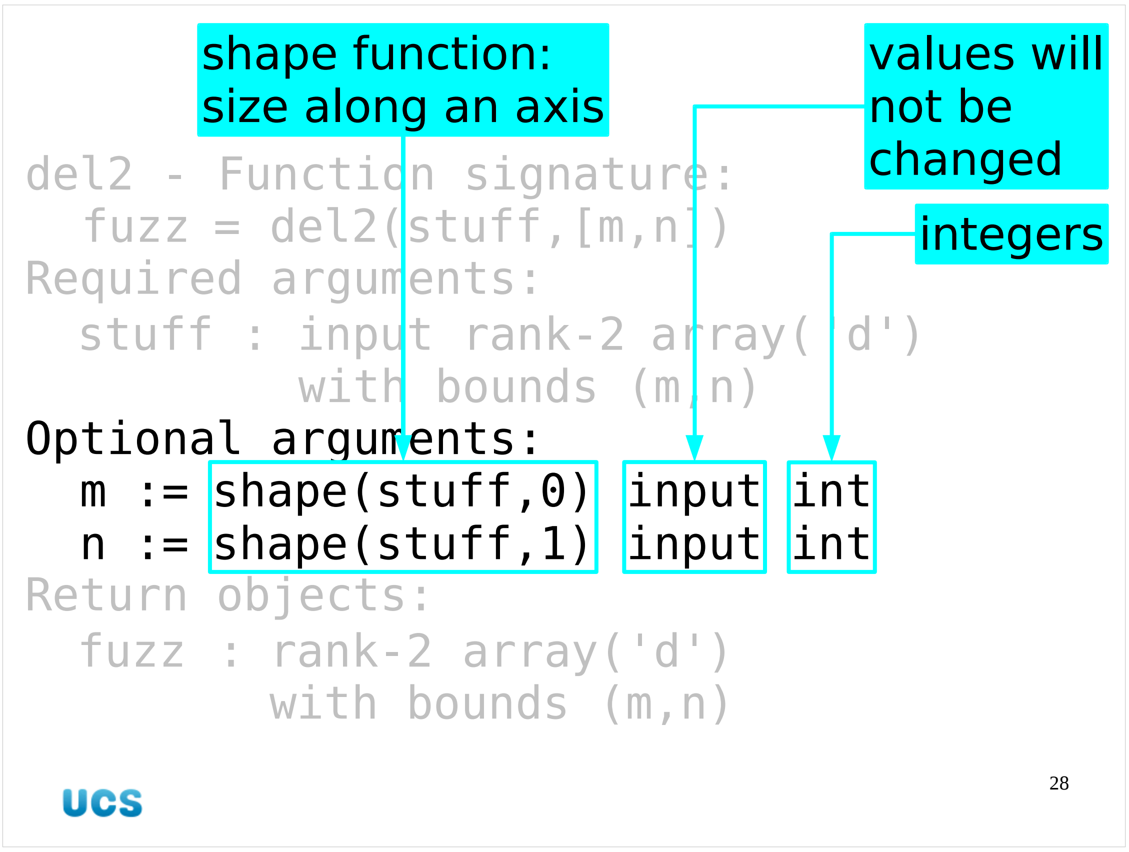
We start with the required argument.

It is described as being an “input” argument. This isn't as self-evident as it appears. Of course it's an input argument, but it's an input-*only* argument. The function will not modify any part of the object being passed in.

It is described as a “rank-2 array” which means it has two axes or dimensions (rows and columns, say). In conjunction with its input-only status this tells us that no element of the array will be modified.

Next we see what it is an array of. (Real numbers, integers, complex numbers,…) The doc strings says it is an array of `'d'`. This is code for “double precision”. The set of data types will be detailed more closely in a later slide, but for now we note that this is an array of double precision numbers.

Finally, we get to see the sizes of the two axes. There are *m* rows and *n* columns. Of course, we don't know what m and n are yet; they're optional arguments. However, this identifies the two letters that will be used elsewhere to describe sizes of arrays in this signature.

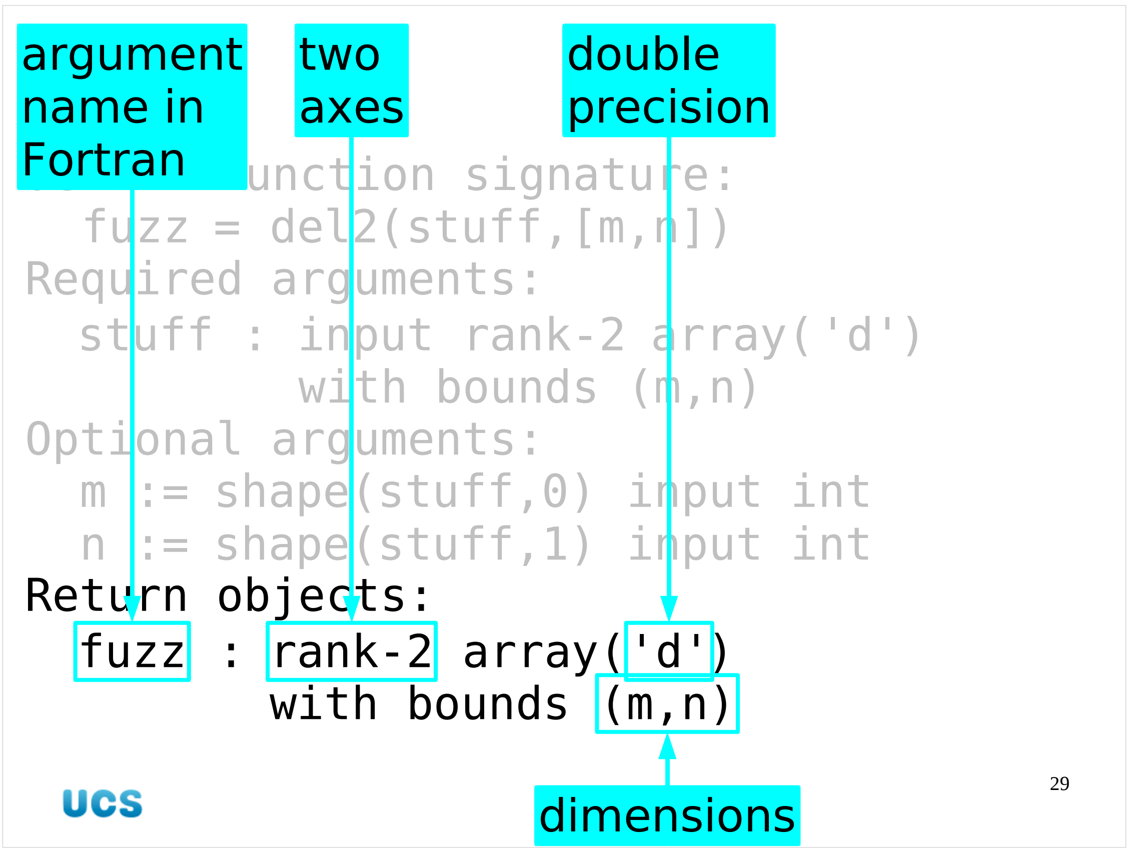


```
del2 - Function signature:
  fuzz = del2(stuff,[m,n])
Required arguments:
  stuff : input rank-2 array('d')
          with bounds (m,n)
Optional arguments:
  m := shape(stuff,0)  input  int
  n := shape(stuff,1)  input  int
Return objects:
  fuzz : rank-2 array('d')
         with bounds (m,n)
```

shape function: size along an axis
values will not be changed
integers
UCS

Next we see the optional arguments. Here again we are told that the values are input-only and that they are integers. But most importantly we see what their default values are. We will see the `shape()` function later, but for now we just need to know that `shape(stuff,0)` is the size of the array along “axis 0”, that is the number of rows. Similarly `shape(stuff, 1)` is the size along “axis 1”, the number of columns.

Given that these “default” values are the only ones we will ever use, we might ask ourselves why the arguments exist at all. We will see how to eliminate them, forcing the use of the `shape()` values, in a few slides' time.

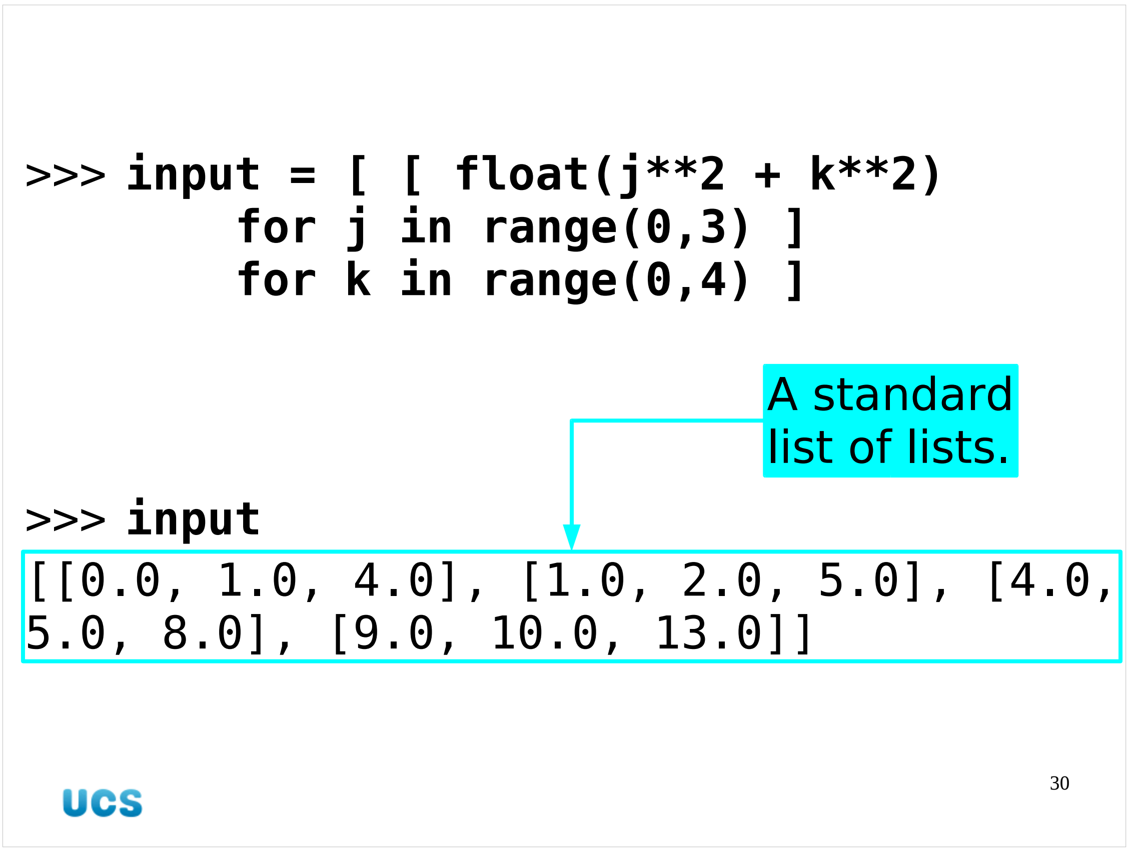

argument name in Fortran
two axes
double precision
dimensions

```
        unction signature:
  fuzz = del2(stuff,[m,n])
Required arguments:
  stuff : input rank-2 array('d')
          with bounds (m,n)
Optional arguments:
  m := shape(stuff,0) input int
  n := shape(stuff,1) input int
Return objects:
  fuzz : rank-2 array('d')
         with bounds (m,n)
```

UCS

29

Finally we see the information about the returned value(s) of the function. Here we see the same definition of the type and shape of the array as we saw for the input array. This time, of course, it is not described as an "input" parameter. Note that by specifying the bounds (m,n) the signature text indicates that the output array is the same size as the input.



```
>>> input = [ [ float(j**2 + k**2)
        for j in range(0,3) ]
        for k in range(0,4) ]

>>> input
[[0.0, 1.0, 4.0], [1.0, 2.0, 5.0], [4.0,
5.0, 8.0], [9.0, 10.0, 13.0]]
```

A standard list of lists.
UCS

Now let's look at what's happening a little more closely.

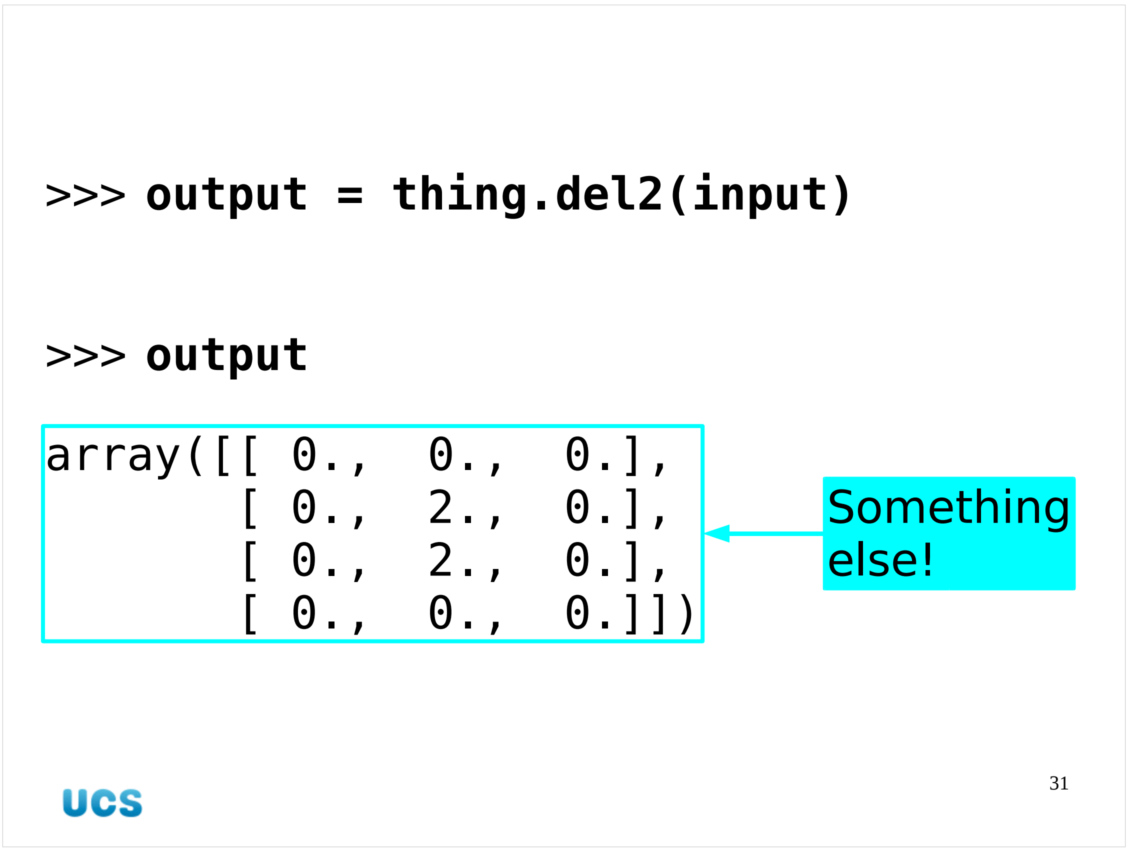
We'll create an array of values to have something to throw at the function. Note that this is just a standard Python list of lists.

We can create the output array by calling the `del2()` function out of the thing module but if we print it out we see that it presents itself rather differently compared to the input array. Whatever it was we got from the `del2()` function, it wasn't a simple list of lists.

Note also that we are exploiting the default values set up for us by the module. We could have run

```
>>> output = thing.del2(input, 4, 3)
```

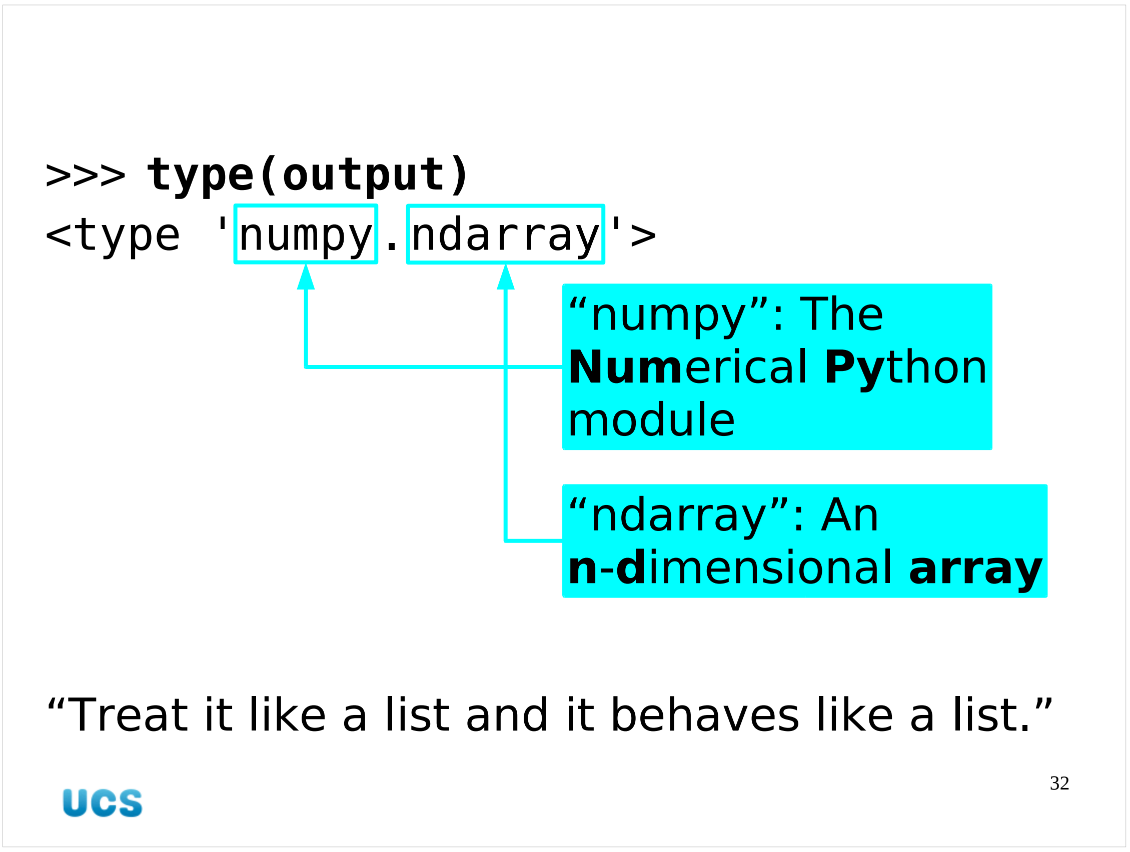
with the same result.



```
>>> output = thing.del2(input)

>>> output
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  0.]])
```

Something else!

So what do we get out of the function? It's not another Pythonic list of lists.



```
>>> type(output)
<type 'numpy.ndarray'>
```

“numpy”: The **Num**erical **Py**thon module
“ndarray”: An **n-d**imensional **array**

“Treat it like a list and it behaves like a list.”

You will recall that Python has a built-in `type()` function which returns the type of an object. We can examine the type of the input and output to confirm that they are, indeed, different.

```
>>> type(input)
<type 'list'>
>>> type(output)
<type 'numpy.ndarray'>
```

The `numpy` module (which we haven't imported explicitly yet but will soon) is the **Num**erical **Py**thon module. Among other things it provides a type of object for storing n-dimensional arrays. It is one of these objects that we have been returned.

But we were able to treat it as if it was just a standard Python “list of lists” array. The `numpy.ndarray` is one of those Python objects designed to behave like a list if you treat it like a list. That's why it worked transparently.

We will return to look at NumPy itself in a few slides' time.

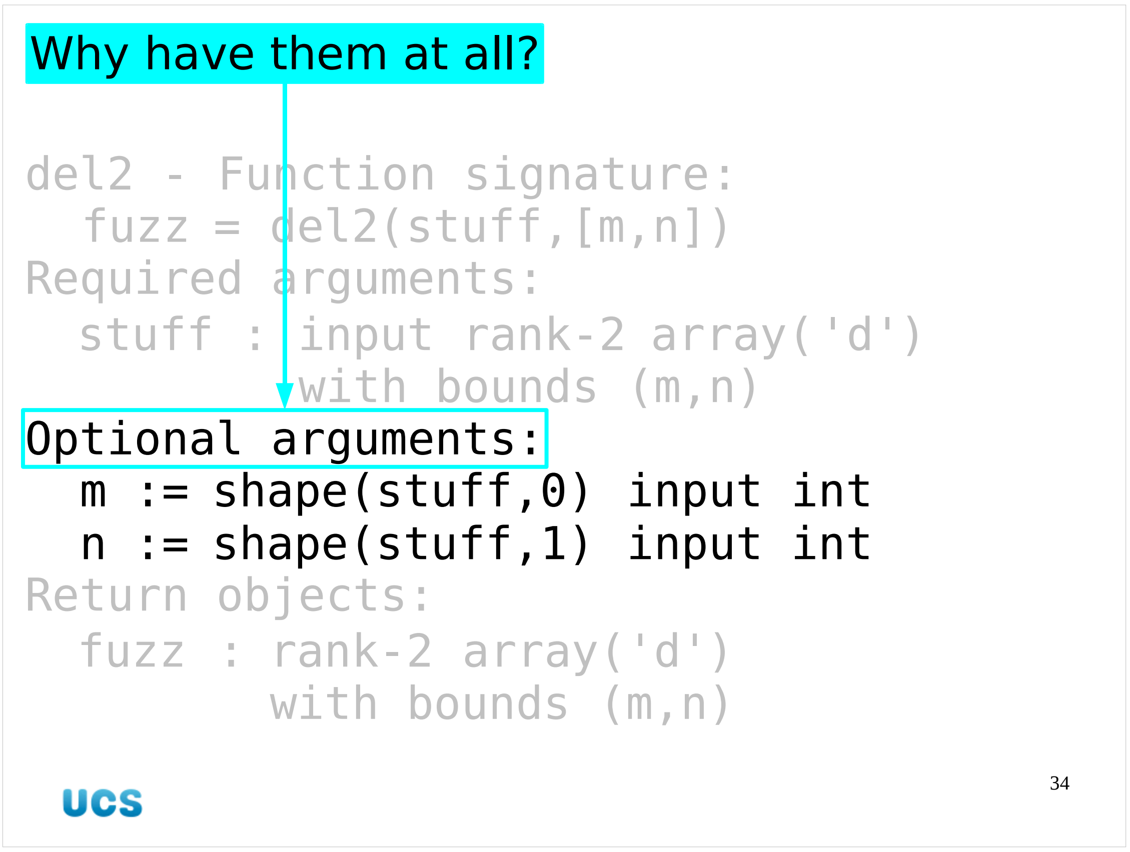
```
>>> output[1,1]
2.0
>>> output[1][1]
2.0
>>> len(output)
4
>>> len(output[0])
3
```

The “Fortran way”

The “Python way”

“Treat it like a list and it behaves like a list.”

This object can be addressed in the traditional Fortran style of a set of indices separated by commas. But while this new type of object is not a normal Python array, we *are* able to treat it as if it was just a standard Python “list of lists” array. The `numpy.ndarray` is one of those Python objects designed to behave like a list if you treat it like a list. That's why it worked transparently.



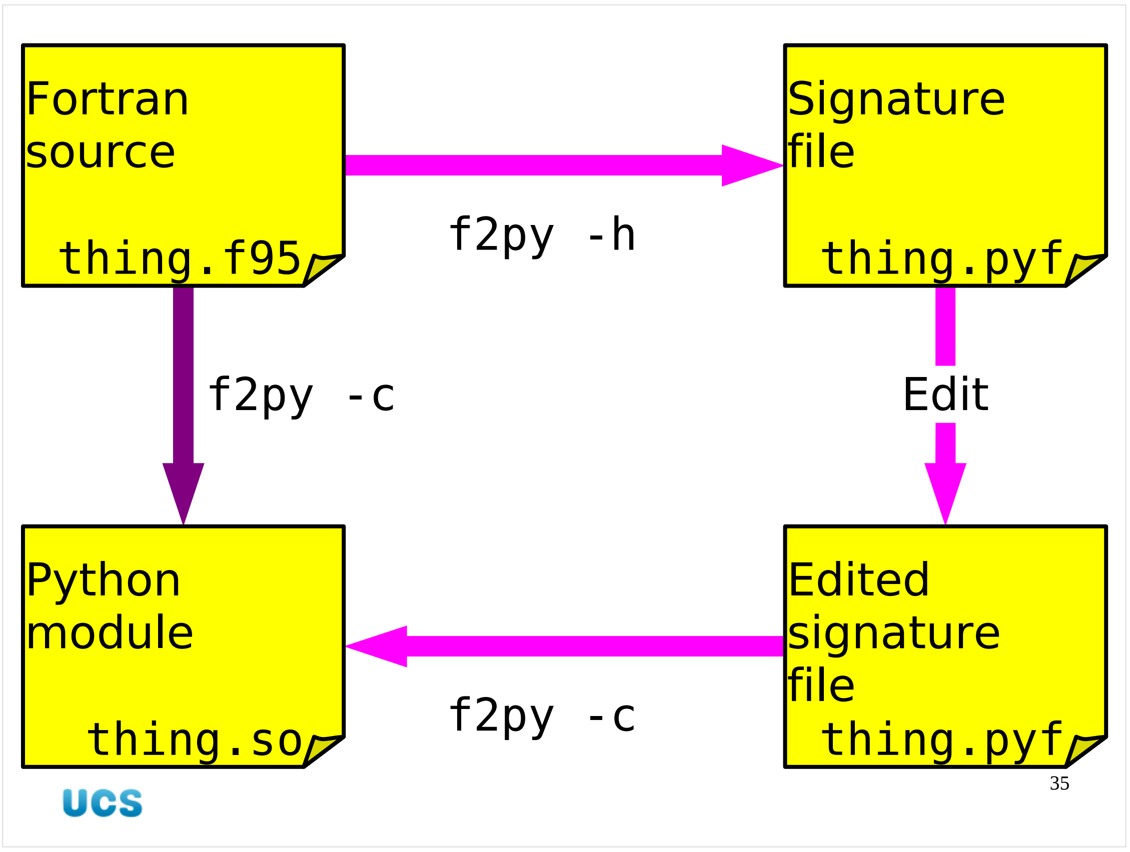
Why have them at all?

```
del2 - Function signature:
  fuzz = del2(stuff,[m,n])
Required arguments:
  stuff : input rank-2 array('d')
          with bounds (m,n)
Optional arguments:
  m := shape(stuff,0) input int
  n := shape(stuff,1) input int
Return objects:
  fuzz : rank-2 array('d')
         with bounds (m,n)
```

Now we will look at cleaning up the interface between Fortran and Python. We currently have some optional arguments whose values, if used, must match those of the array. Why should we have them at all?

Next we will examine how to change the automatically provided function signatures given to us by f2py.

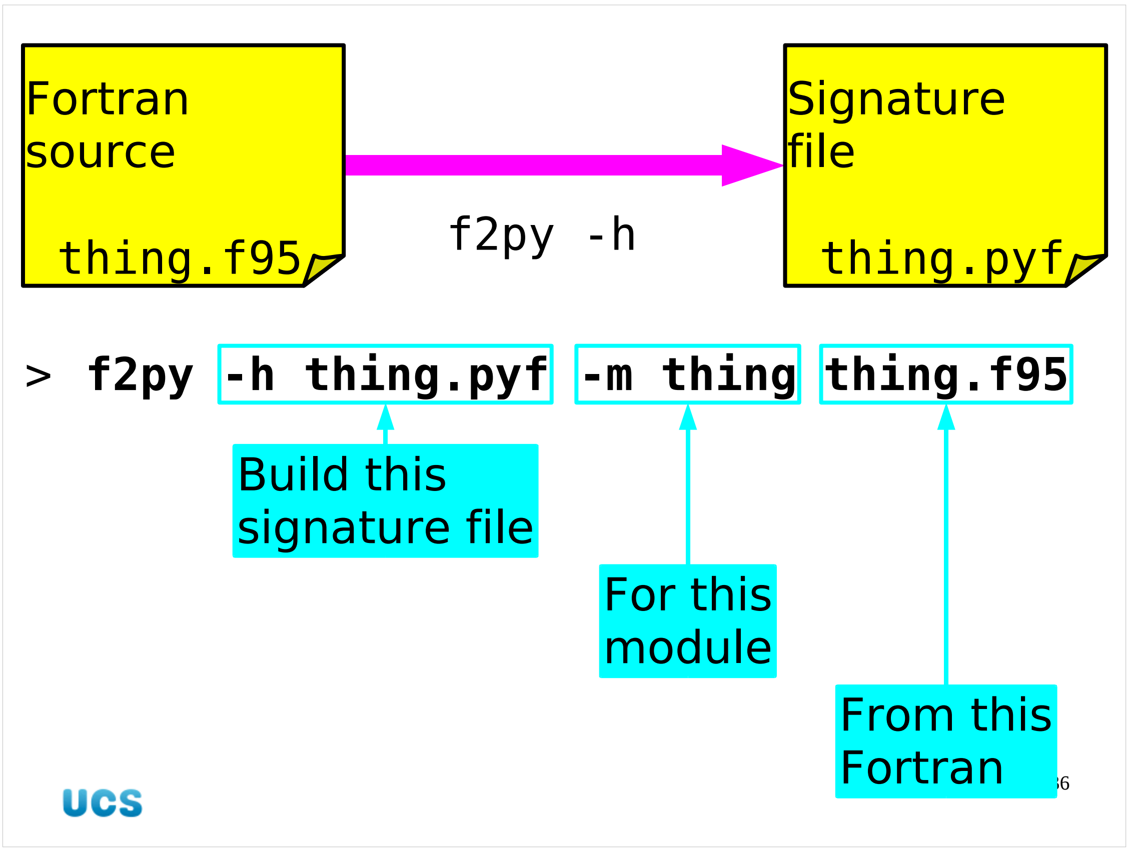
We will also be able to use this technique to get Fortran 77 programs to work for us.



At the moment we simply convert directly from the Fortran 95 file to the compiled Python module.

What we are going to do is to take a more circuitous route, getting to see and modify a file which only appears internally when we go direct. This is called the “signature file” and determines the interfaces of the functions in the module created.

We will start by not changing the signature at all, to make sure that we can recreate what we already have.

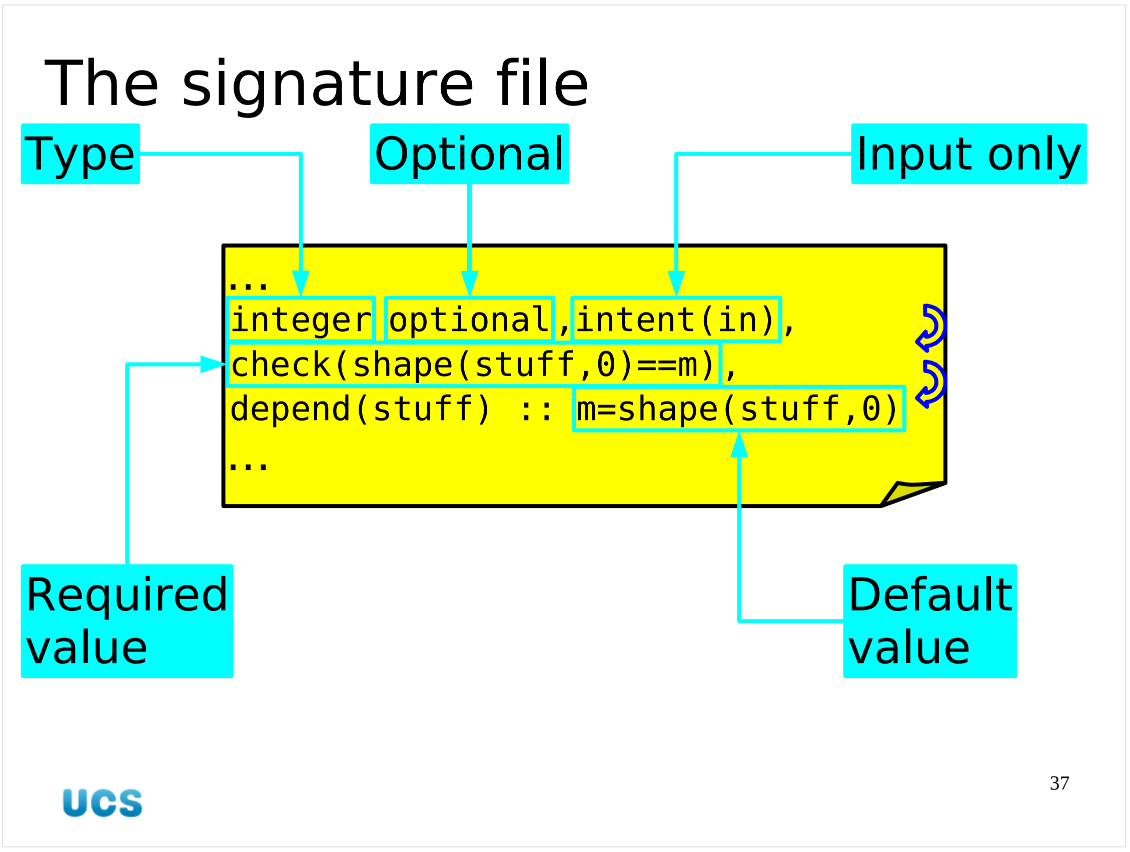


```
> f2py -h thing.pyf -m thing thing.f95
```

Initially we take our f2py command and pass it options to tell it to build a signature file (`-h thing.pyf`) rather than to compile a module (`-c`). Because we aren't compiling anything we don't need to tell it our compiler (`--fcompiler=`…). Apart from that the command is the same as usual:

```
> f2py -h thing.pyf -m thing thing.f95
```

We get a file called “`thing.pyf`” created. This is the signature file.



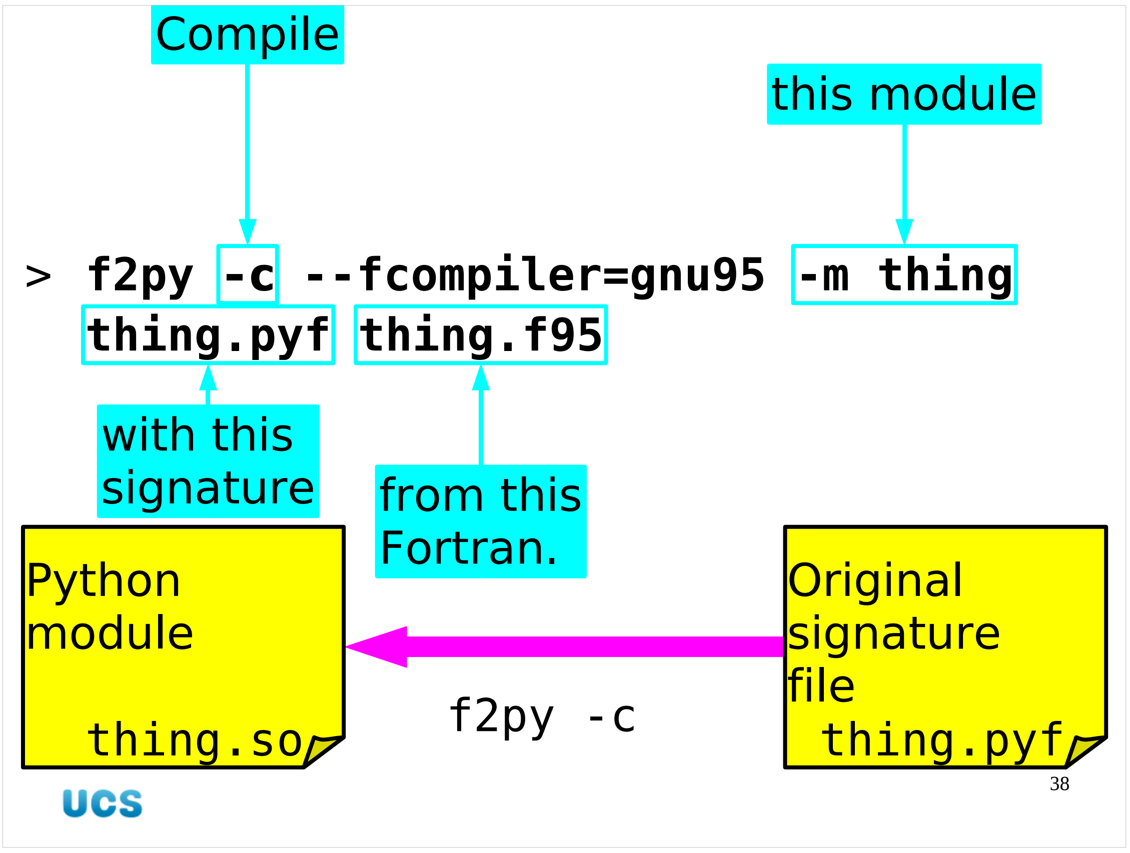
The signature file

Type
Optional
Input only
...
integer optional,intent(in),
check(shape(stuff,0)==m),
depend(stuff) :: m=shape(stuff,0)
...
Required value
Default value
37
UCS

We do not need to understand the syntax of the signature file in any depth. We can follow our noses and achieve enough. We will focus on the lines corresponding to the arguments of the subroutine.

The signature file in its entirety should look something like this. Note that it uses an exclamation mark, “!”, to introduce comments.

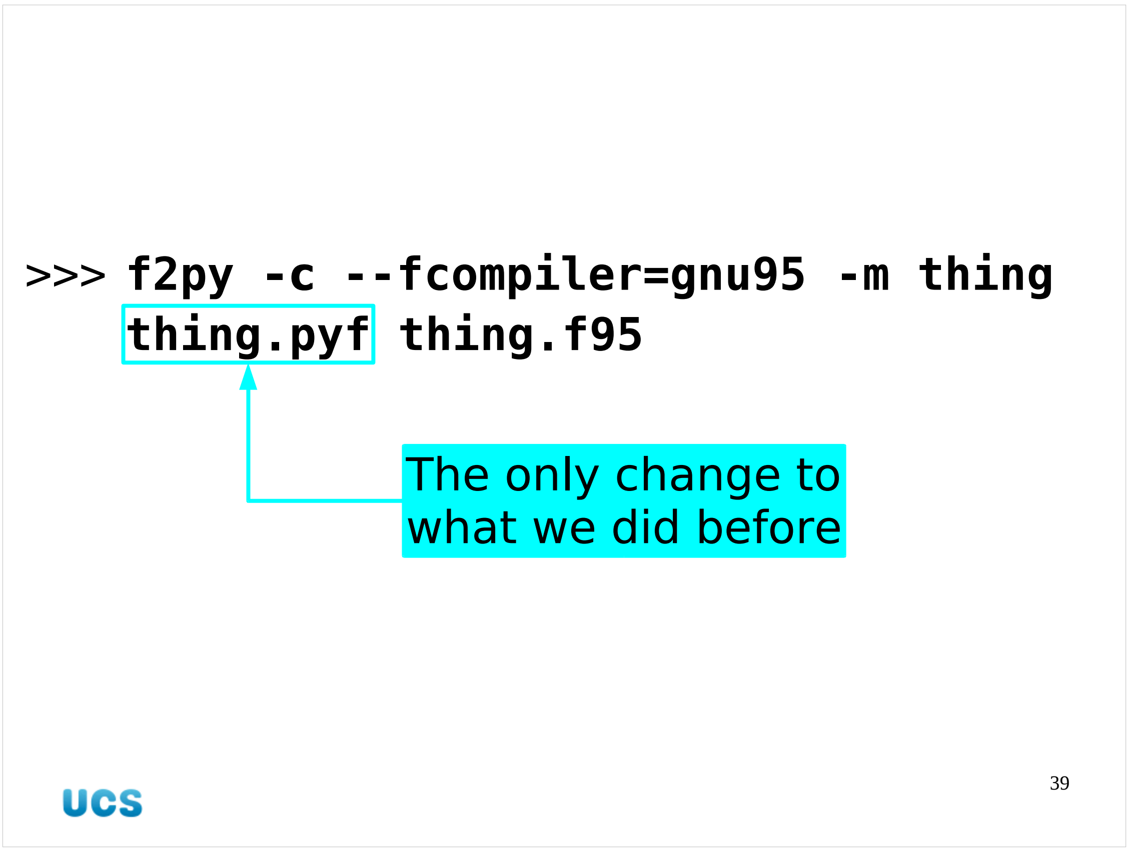
```
python module thing ! in
    interface  ! in :thing
        subroutine del2(array,fuzz,m,n) ! in :thing:thing.f95
            double precision dimension(m,n), intent(in) :: array
            double precision dimension(m,n), intent(out), 
depend(m,n) :: fuzz
            integer optional, intent(in), 
check(shape(array,0)==m), depend(array) :: m=shape(array,0)
            integer optional, intent(in), 
check(shape(array,1)==n), depend(array) :: n=shape(array,1)
        end subroutine del2
    end interface
end python module thing
```



Now we will complete the compilation process, using the signature file. Of course, this won't gain us anything because we haven't changed that file.

```
> f2py -c --fcompiler=gnu95 -m thing thing.pyf thing.f95
```

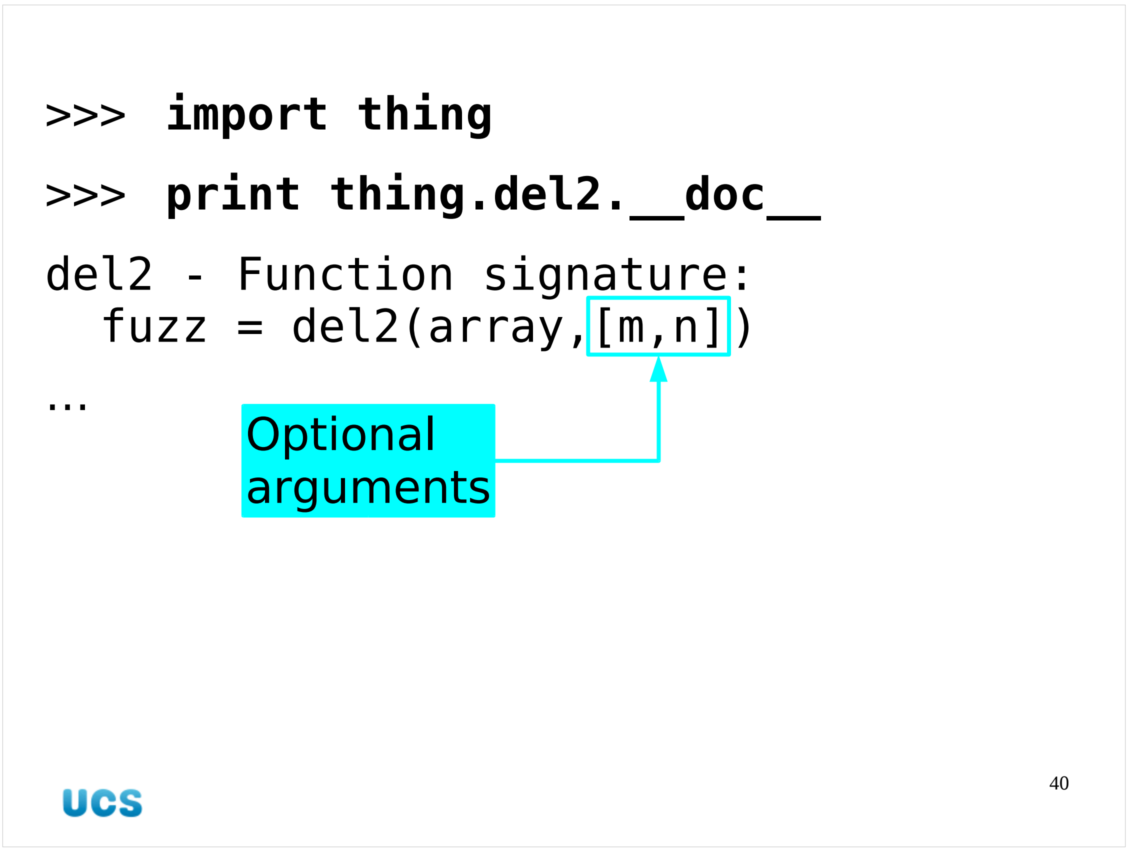
This produces a shared object Python module which is essentially identical to the one we got from going direct. (They're not byte-for-byte identical but their functions are the same.)



```
>>> f2py -c --fcompiler=gnu95 -m thing
    thing.pyf thing.f95
```

The only change to what we did before

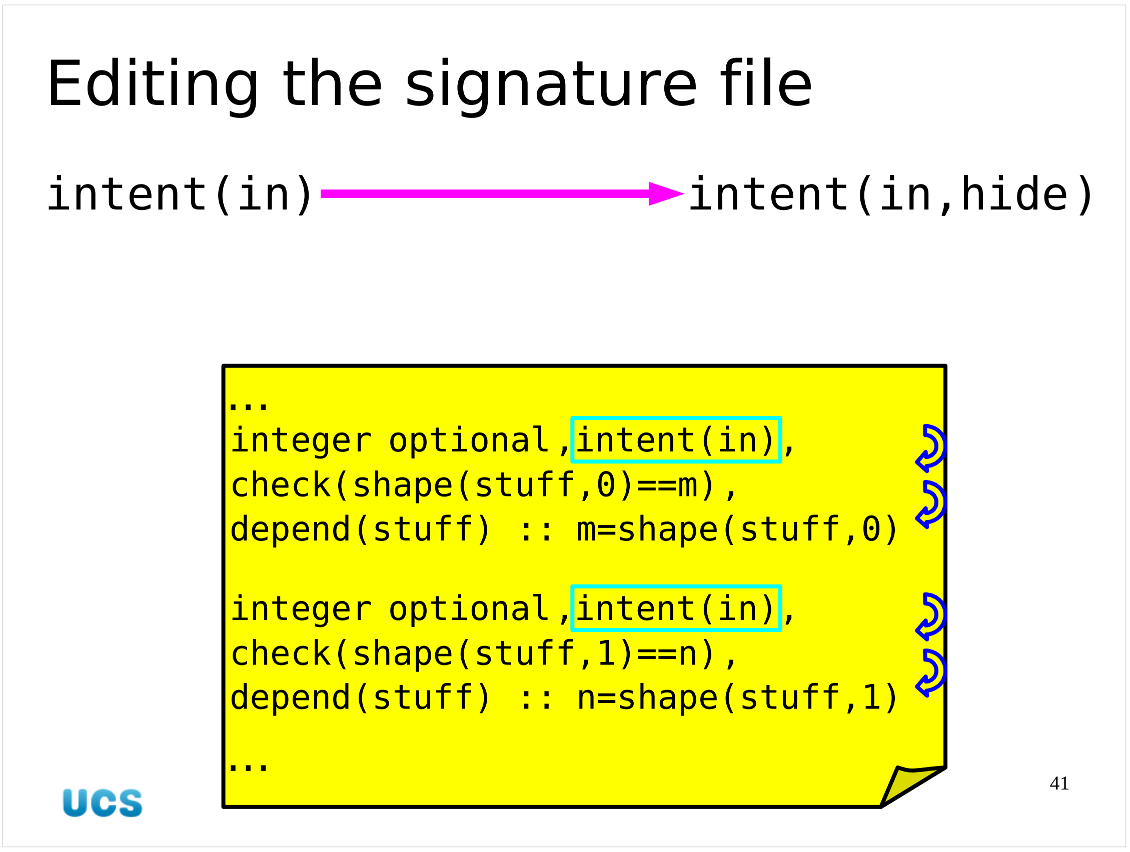
The command is identical to our previous all-in-one compilation command except that the source code files are preceded by the (potentially edited) signature file.



```
>>> import thing
>>> print thing.del2.__doc__
del2 - Function signature:
  fuzz = del2(array,[m,n])
…
```

Optional arguments

We can check the function signatures to see they are totally unchanged.



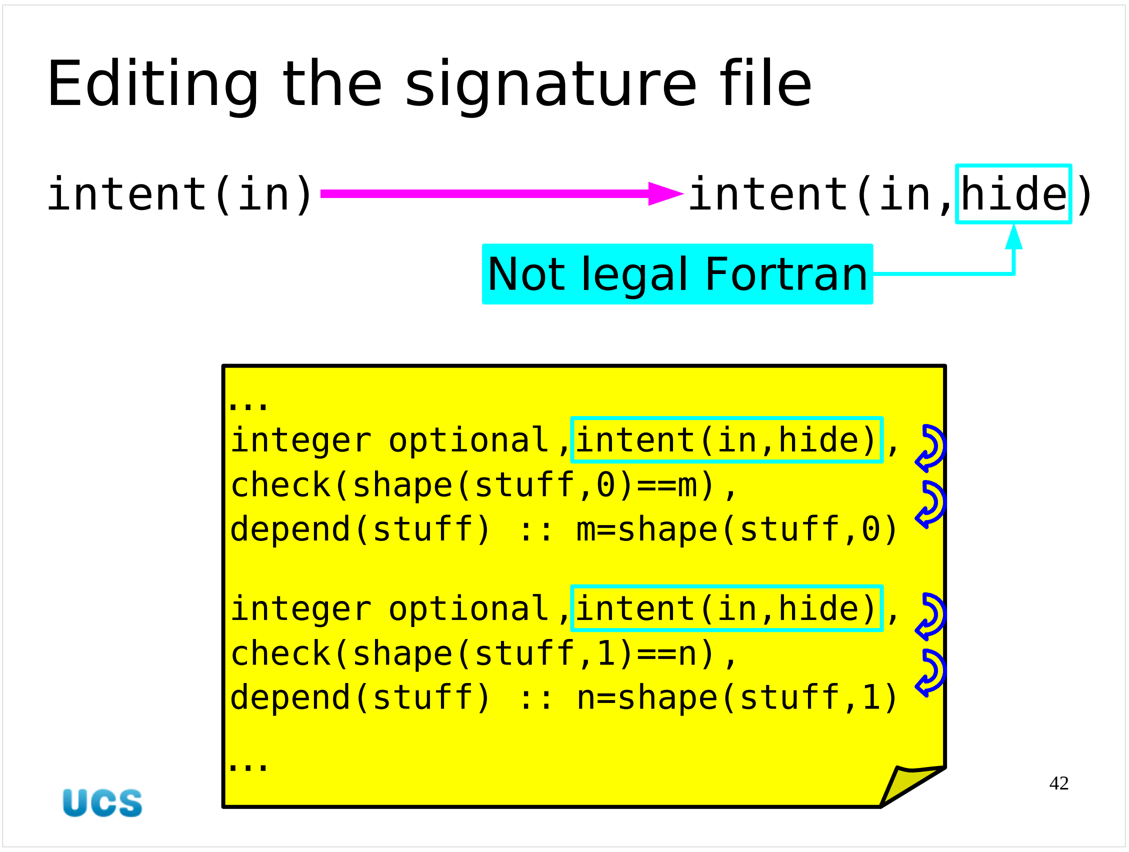
Editing the signature file

$\texttt{intent(in)} \xrightarrow{\hspace{3cm}} \texttt{intent(in,hide)}$

```
...
integer optional,intent(in),
check(shape(stuff,0)==m),
depend(stuff) :: m=shape(stuff,0)

integer optional,intent(in),
check(shape(stuff,1)==n),
depend(stuff) :: n=shape(stuff,1)
...
```

So let's look at the signature file. We do not need to know its entire syntax, just a couple of simple tricks.

We will concentrate on the lines that relate to the two optional arguments, *m* and *n* and, in particular, the "`intent()`" statement. This is derived directly from the Fortran source. In a Python signature file, however, we can add intentions that are not legal Fortran. We adjust the text "intent(in)" to "intent(in,hide)". This is not legal Fortran — there is no `hide` option — but is legal in a Python signature file.



Editing the signature file

`intent(in)` $\rightarrow$ `intent(in,hide)`

Not legal Fortran

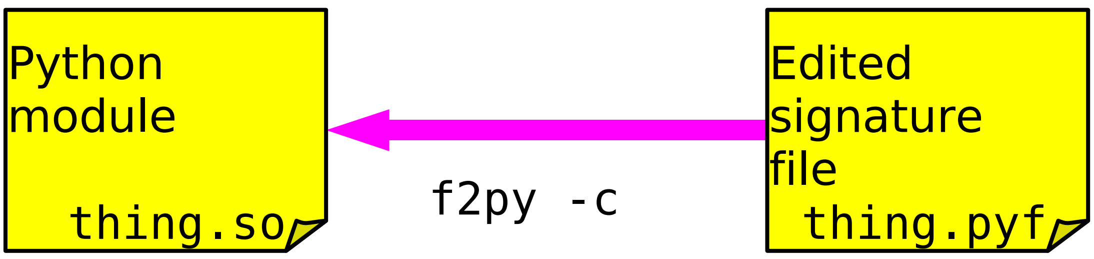
```
...
integer optional,intent(in,hide),
check(shape(stuff,0)==m),
depend(stuff) :: m=shape(stuff,0)

integer optional,intent(in,hide),
check(shape(stuff,1)==n),
depend(stuff) :: n=shape(stuff,1)

...
```

```
python module thing ! in
    interface  ! in :thing
        subroutine del2(array,fuzz,m,n) ! in :thing:thing.f95
            double precision dimension(m,n), intent(in) :: array
            double precision dimension(m,n), intent(out),
depend(m,n) :: fuzz
            integer optional, intent(in,hide),
check(shape(array,0)==m), depend(array) :: m=shape(array,0)
            integer optional, intent(in,hide),
check(shape(array,1)==n), depend(array) :: n=shape(array,1)
        end subroutine del2
    end interface
end python module thing
```

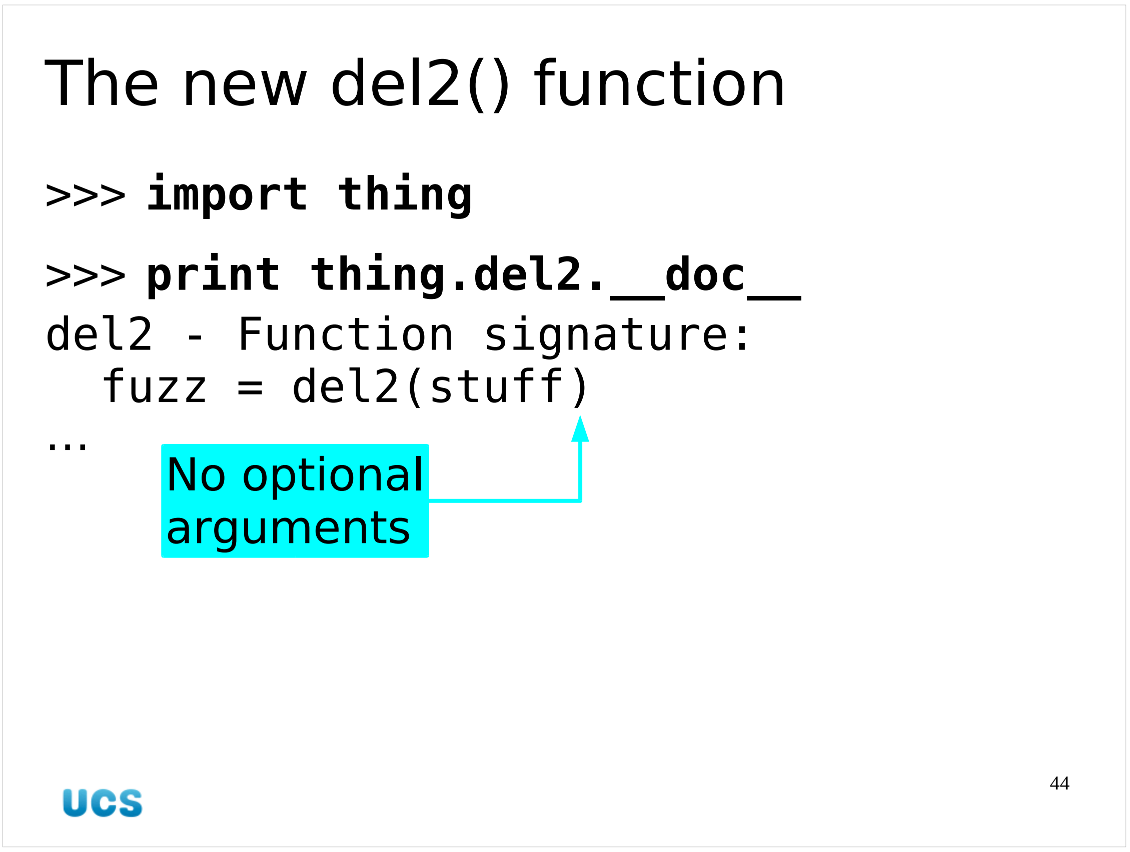
Using the signature file

```
>>> f2py -c --fcompiler=gnu95 -m thing
    thing.pyf thing.f95
```

"intent(in,hide)"



Now we will rebuild the Python module using our modified signature file.



The new del2() function

```
>>> import thing
>>> print thing.del2.__doc__
del2 - Function signature:
  fuzz = del2(stuff)
...
```

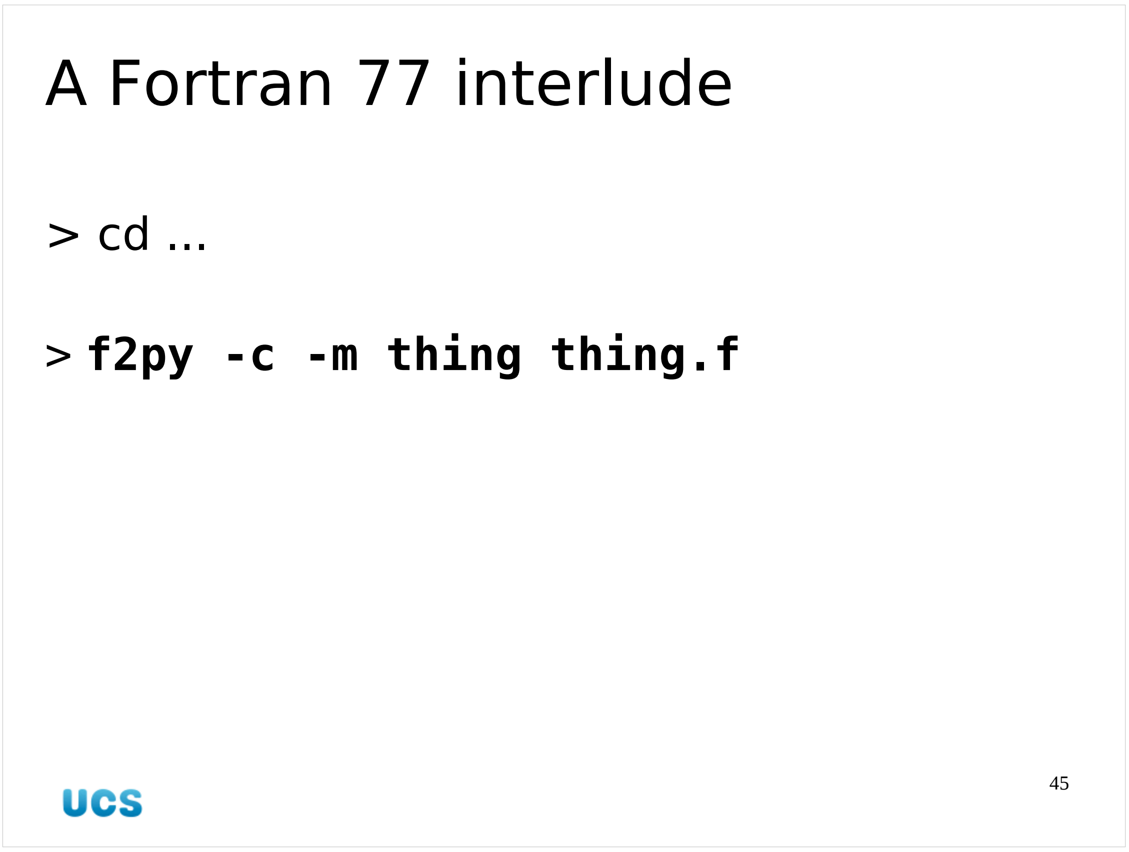
No optional arguments

And now we see a new function signature from the Python module with no mention of the optional arguments and a far more “Pythonic” style of interface.

Is it worth doing?

If you are creating a “quickie” Python interface for just your own quick use then almost certainly not.

If you are creating a module for use and re-use, possibly by other people who know less about the Fortran than you do, then almost certainly it is worth it.



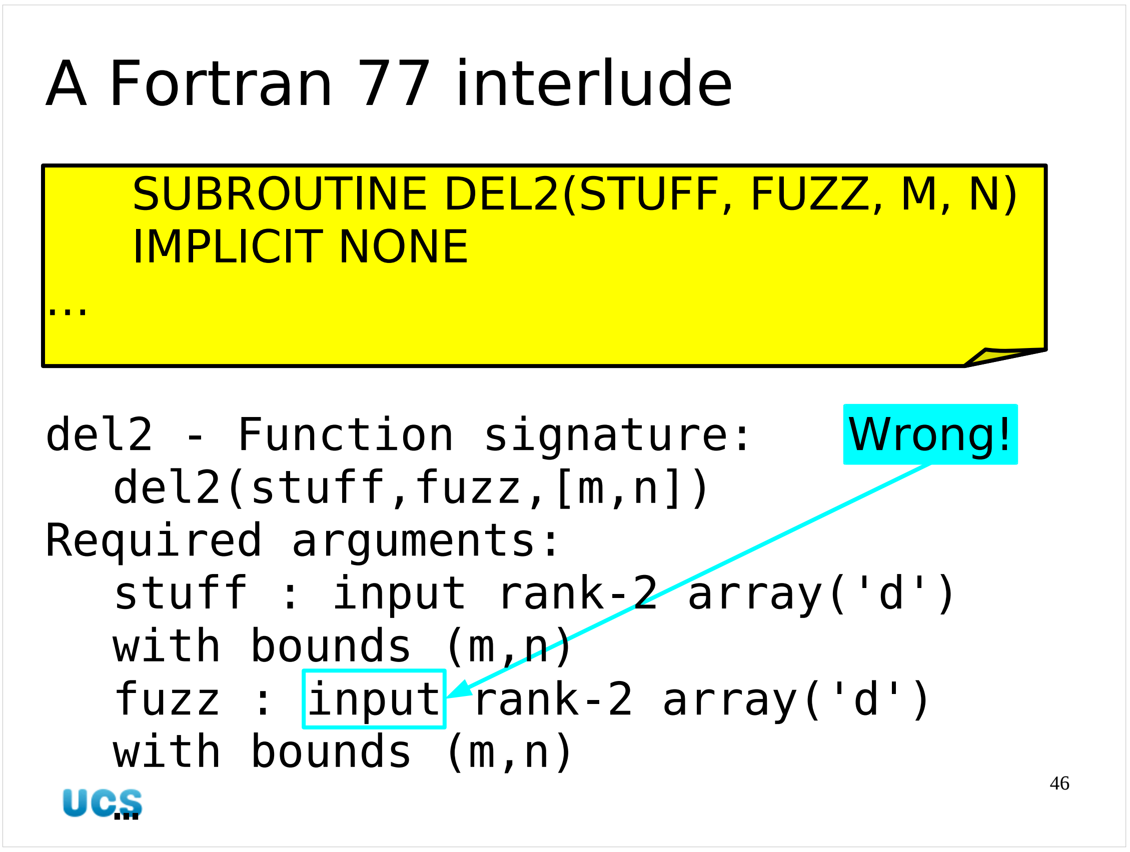
A Fortran 77 interlude

```
> cd ...

> f2py -c -m thing thing.f
```

There is one other reason for introducing signature files. They are one route towards integrating Fortran 77 to Python.

If you aren't interested in Fortran 77 you can stop listening for a bit. First we change directory to ...

Once there we build a Python module just as we have in the past.



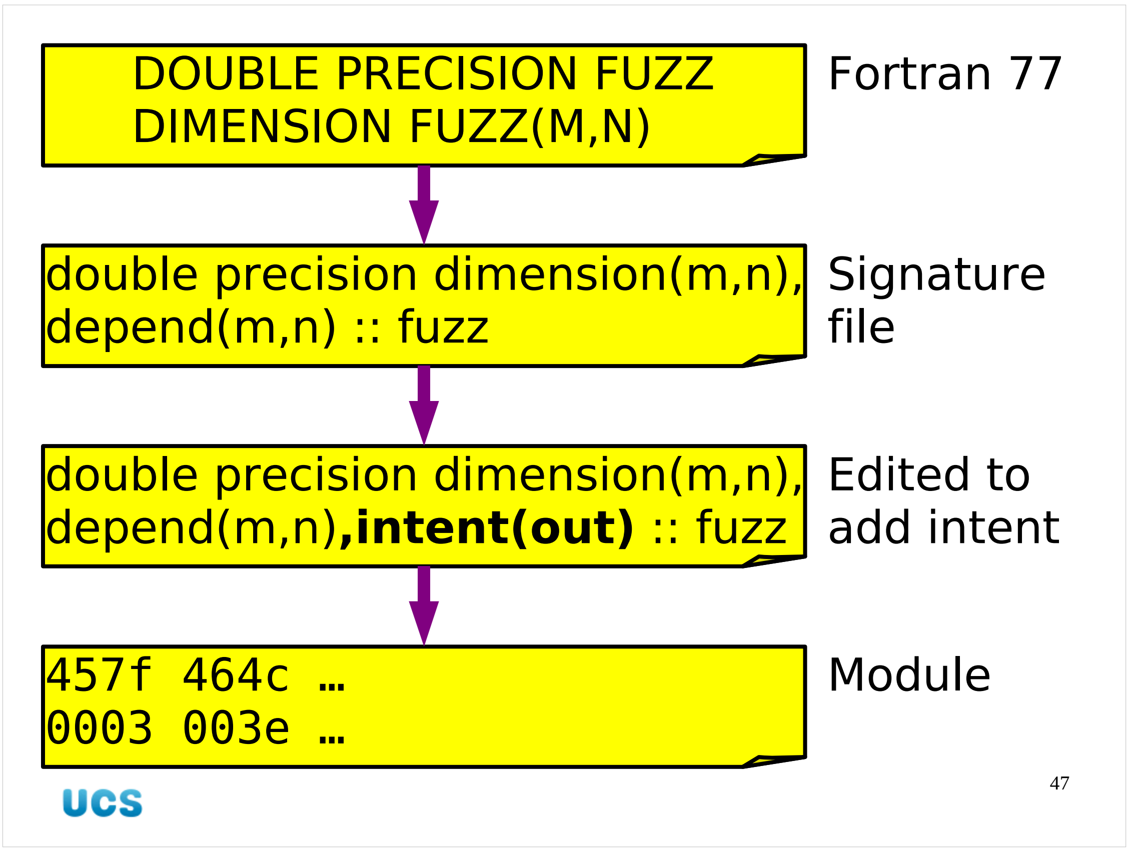
A Fortran 77 interlude

```
      SUBROUTINE DEL2(STUFF, FUZZ, M, N)
      IMPLICIT NONE
…
```

```
del2 - Function signature:
  del2(stuff,fuzz,[m,n])
Required arguments:
  stuff : input rank-2 array('d')
  with bounds (m,n)
  fuzz : input rank-2 array('d')
  with bounds (m,n)
```

Wrong!

We observe that the function is not what we want. The Python function has been created with what was out return value as a function argument. We need to know how to adjust this.



```
      DOUBLE PRECISION FUZZ
      DIMENSION FUZZ(M,N)
```

Fortran 77

```
double precision dimension(m,n),
depend(m,n) :: fuzz
```

Signature file

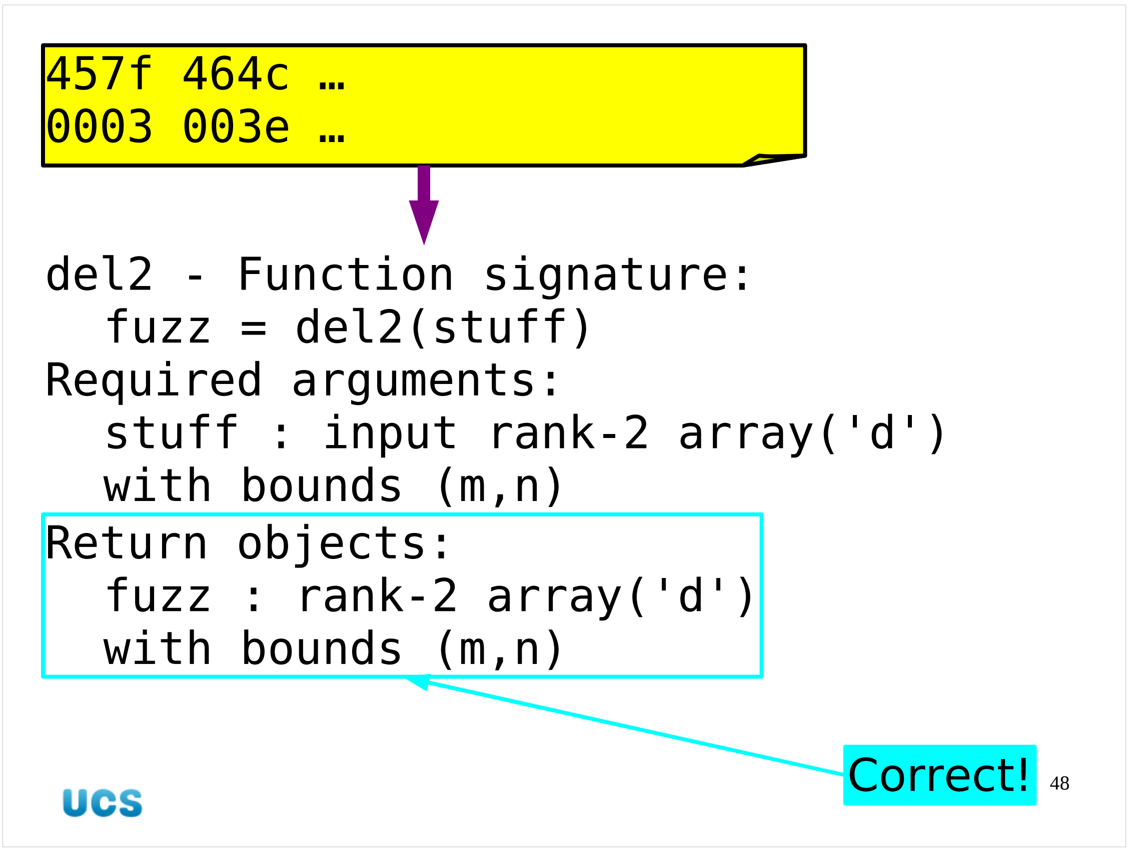
double precision dimension(m,n),
depend(m,n)**,intent(out)** :: fuzz

Edited to add intent

```
457f 464c …
0003 003e …
```

Module

Instead of moving directly from the Fortran 77 source to the Python module we will create the signature file first. Once we have it we will edit it to specify that the Fortran argument is intended for outgoing values only. Then we use the edited signature file to generate the final module.



```
457f 464c …
0003 003e …
```

```
del2 - Function signature:
  fuzz = del2(stuff)
Required arguments:
  stuff : input rank-2 array('d')
  with bounds (m,n)
Return objects:
  fuzz : rank-2 array('d')
  with bounds (m,n)
```

Correct!
UCS

And now we find that the function has the signature we wanted.

If we had wanted to we could have added `intent(in,hide)` for m and n too.

The NumPy module

Part of the Scientific Python suite of modules.

Provides the fundamental n-dimensional arrays

(And some other stuff…)

http://www.scipy.org/NumPy/

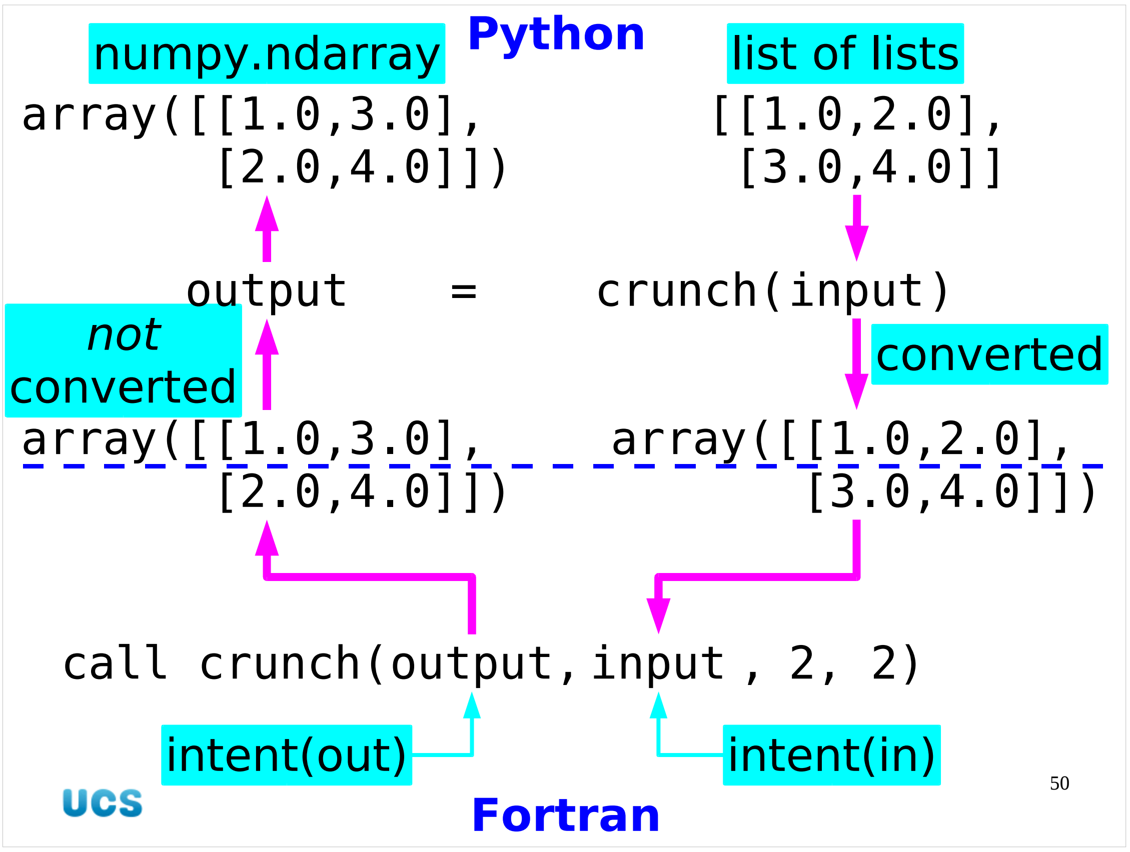
http://www.scipy.org/Topical_Software/

And now back to NumPy.

On-line documentation about the module itself can be found at `http://www.scipy.org/NumPy/` but the documentation on the `f2py` tool is somewhat lacking at present.

It lies at the core of a set of modules known as “Scientific Python”. There are far more modules in that set than we can cover in one afternoon, and you should visit their web site for current information. A subject-oriented guide to the set of modules can be found at `http://www.scipy.org/Topical_Software` and should be your starting place.

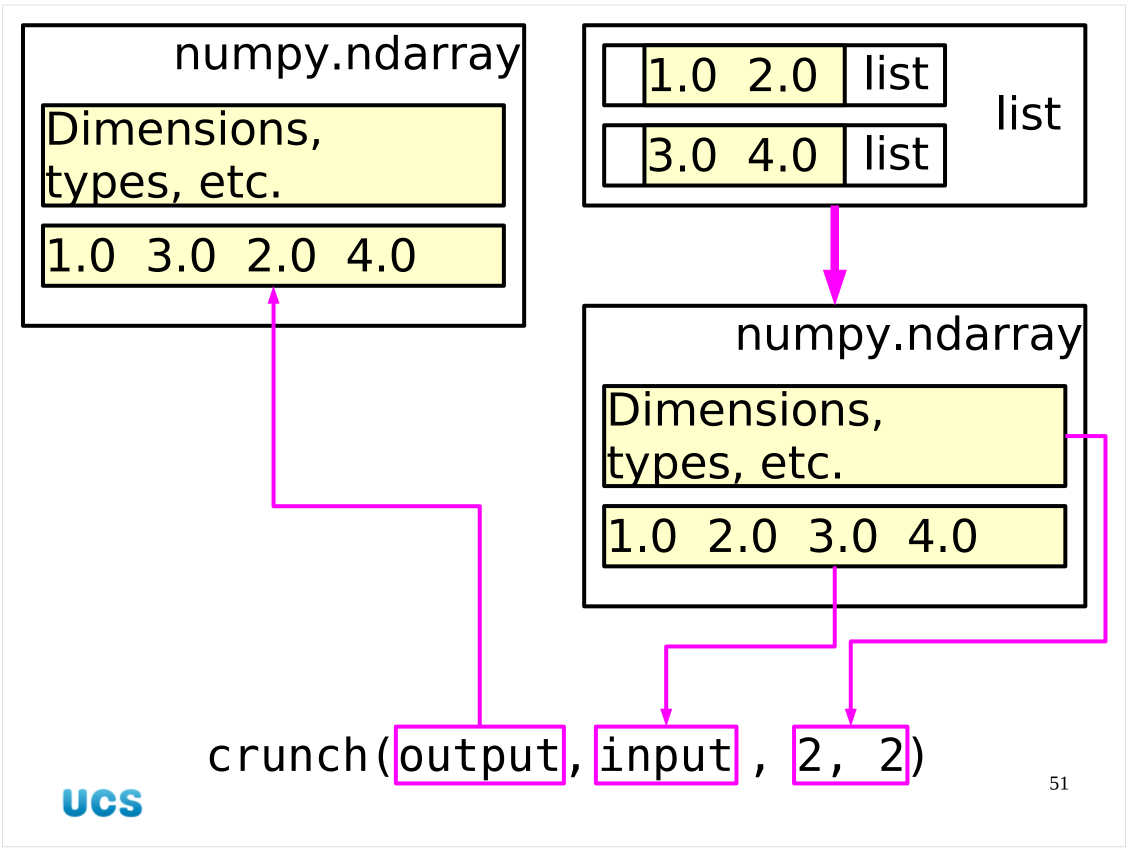
In this afternoon's course we will focus on the aspects of `numpy` relevant to `f2py`.



Let's work out how we got from the list of lists we started with for our array and ended up with the NumPy array our function gave us back.

When we fed our list of lists to our function the module created by f2py automatically converted it to a NumPy array. People who require peak performance should notice that that's an operation that will be run every time we feed in a pure Python list of lists, as part of a loop say. If we create a NumPy array in Python (we'll see how to later) and pass it in there's no conversion, obviously.

Recall that we wrote the procedure in Fortran as a subroutine, not a function. It gets passed the array to fill in. Again there is scope for inefficiency if it has to do this time and time again for something which could be the same object.



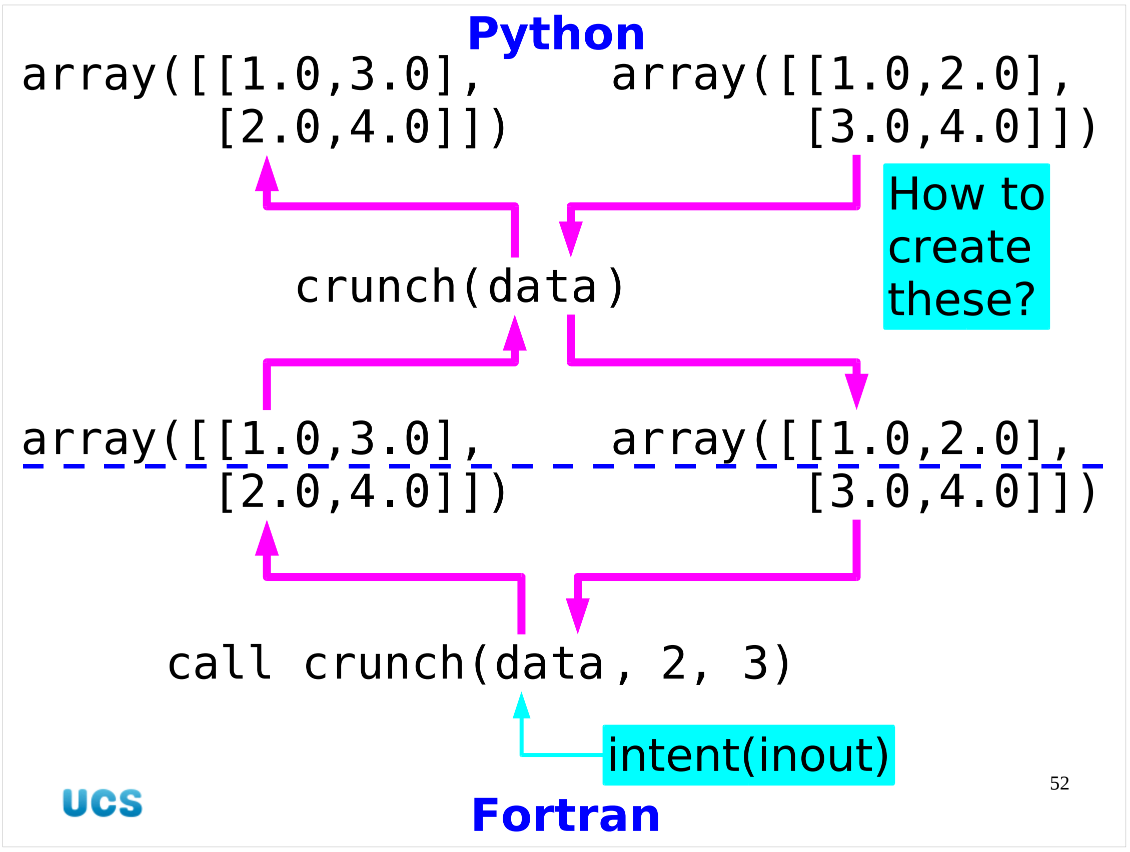
So how does it work?

NumPy arrays are genuine Python objects that store their data internally in one of the layouts as C or Fortran uses. (We'll see how to select which in a moment.) When used from Python the whole object is addressed. When used from C or Fortran the memory address of just that inner data-bearing core is passed to the system.

So when we fed our list of lists to our function the module created by f2py automatically converted it to a NumPy array. This has Fortran-addressable data within it alongside the extra data that Python uses.

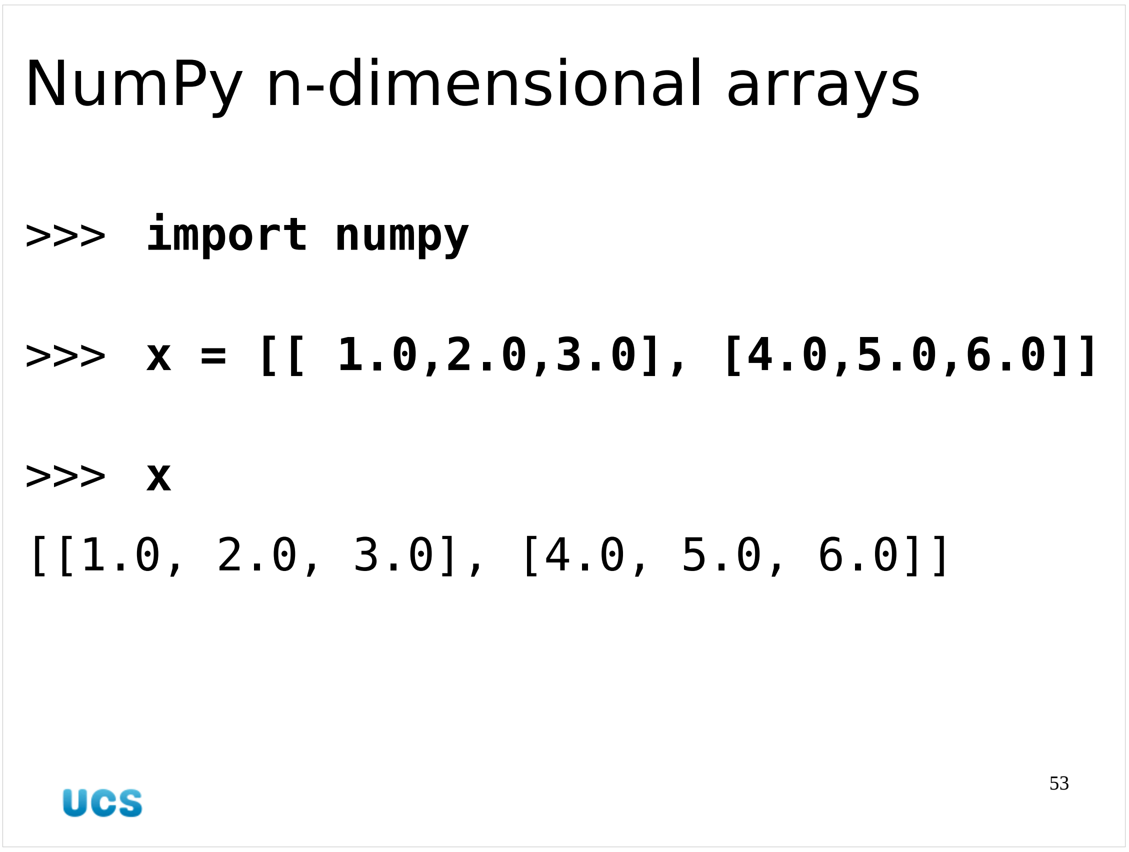
Because we wrote the procedure in Fortran as a subroutine, not a function, it gets passed the already-existing array to fill in. That array is created by the f2py wrappings as another NumPy array and the address of the Fortran-style data layout is passed to the Fortran we wrote.

When the output object is passed back to the Python the Python system picks up the “wrapping” Python object.



We do need to know how to create NumPy arrays within Python; we can't just rely on our f2py-created modules to do magic.

Suppose we were using a Fortran subroutine that manipulated a data array in place. We have to be able to pass it in in the right form already.



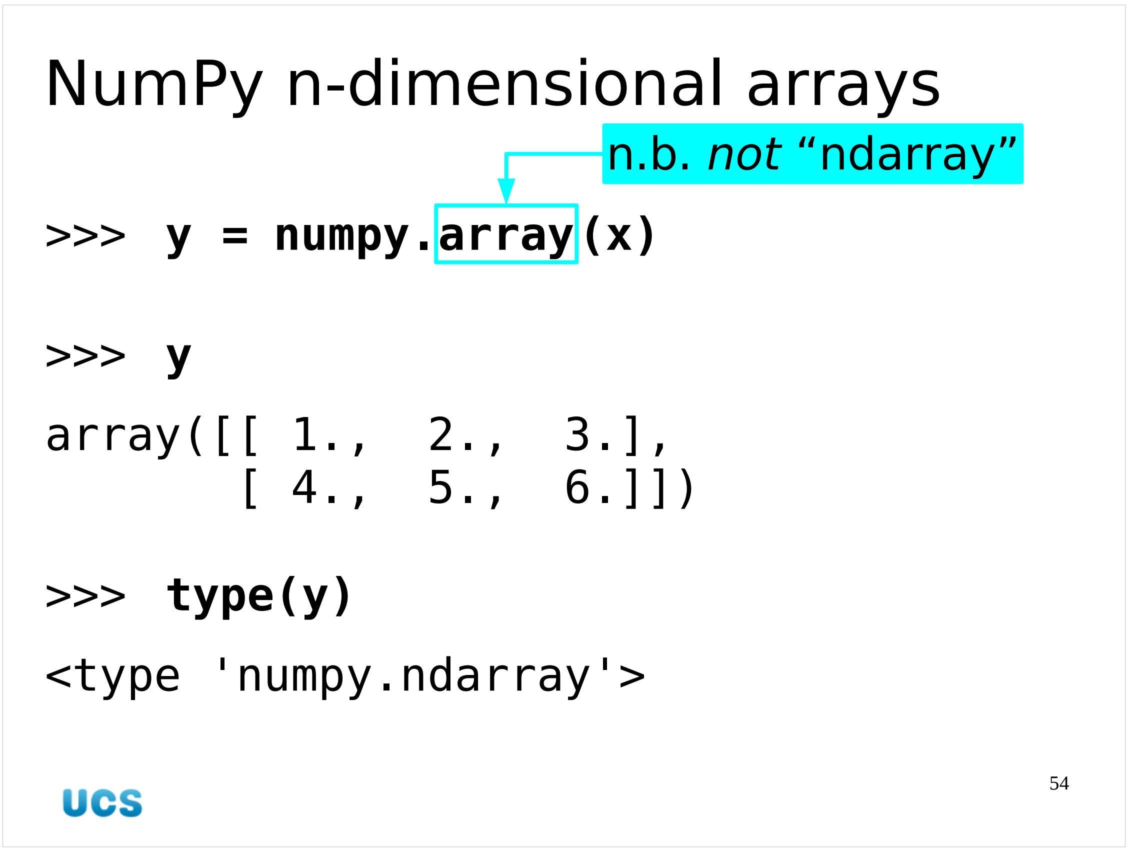
NumPy n-dimensional arrays

```
>>>  import numpy

>>>  x = [[ 1.0,2.0,3.0], [4.0,5.0,6.0]]

>>>  x
[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
```

So let's look at the NumPy module itself.

We'll start by importing the num,py module. Note that the module name is all lower case.

For our data we will use a fairly trivial 2×3 array represented in Python style as a list of two lists of three elements.



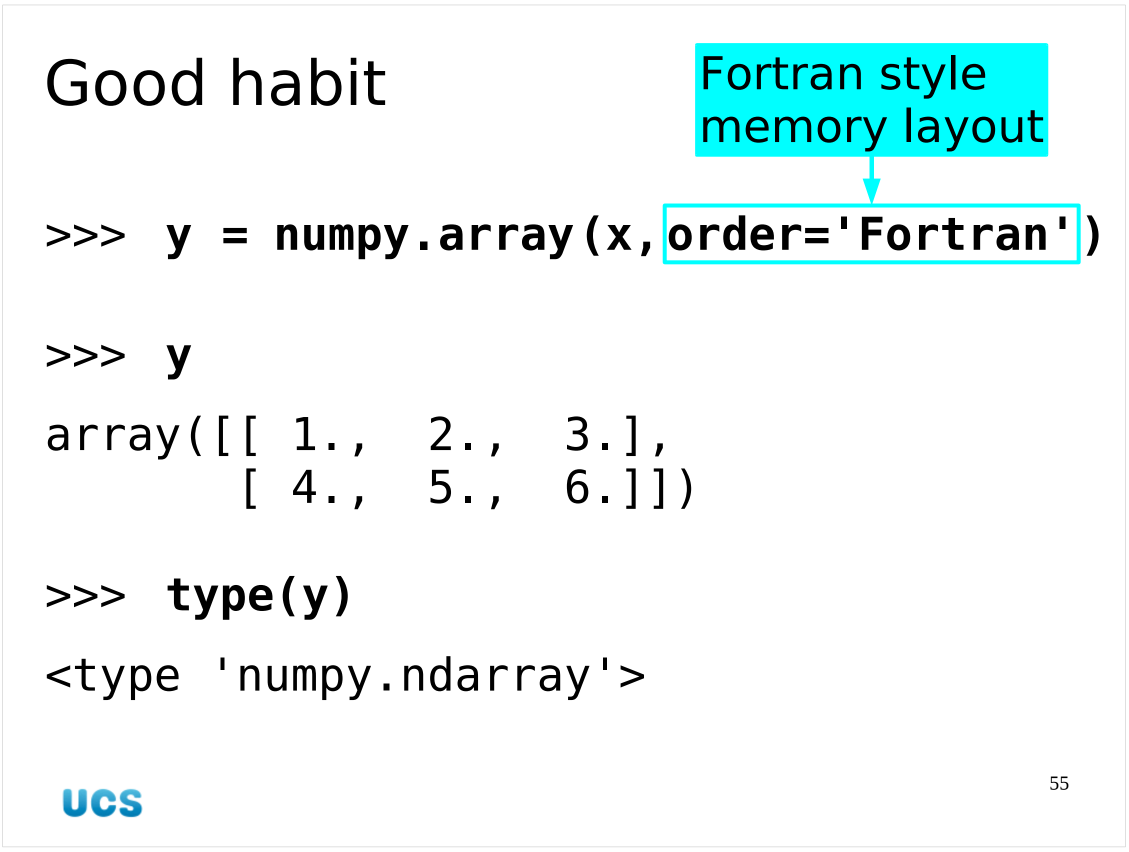
NumPy n-dimensional arrays

n.b. *not* “ndarray”

```
>>> y = numpy.array(x)

>>> y
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

>>> type(y)
<type 'numpy.ndarray'>
```

Now we create a NumPy array.

The numpy module has a function array() which creates a numpy.ndarray object from a Python list. Note that the function is called just “array()” and not “ndarray()”. (In object-oriented speak this is a factory function, not a constructor.)

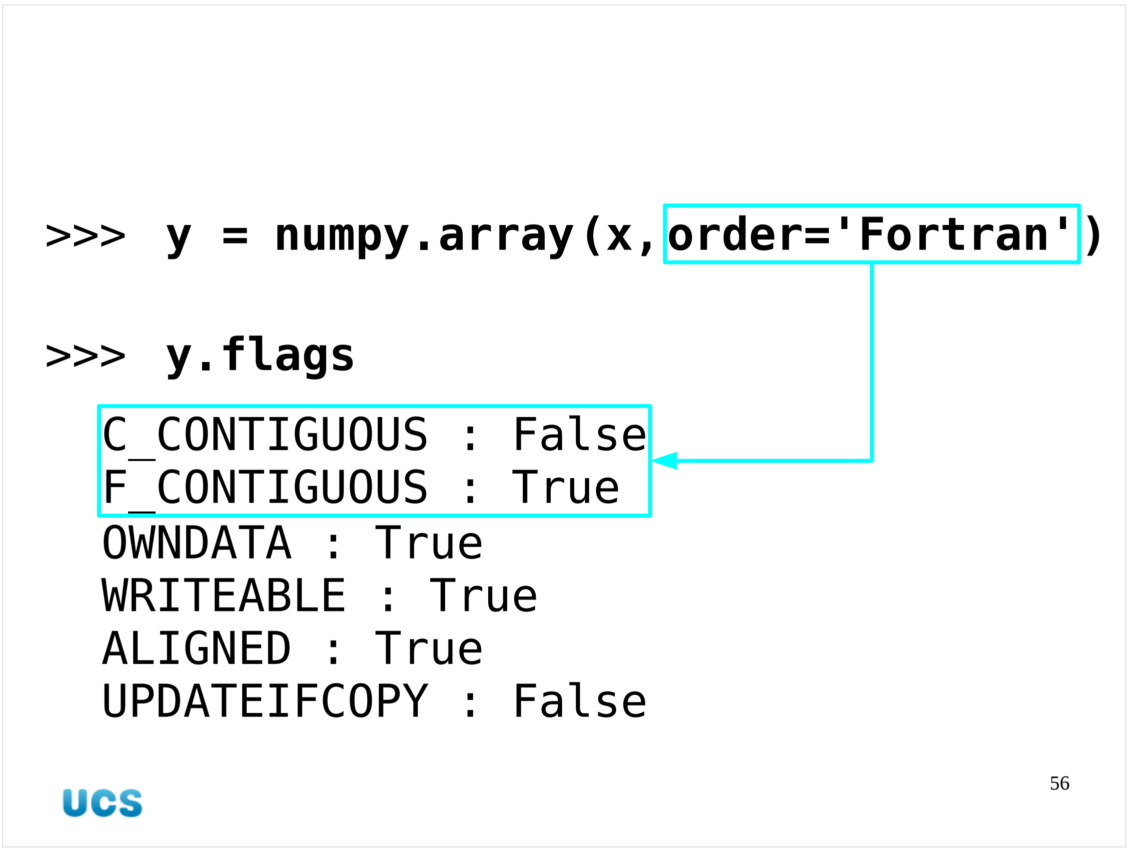


Good habit

Fortran style memory layout

```
>>> y = numpy.array(x,order='Fortran')

>>> y

array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

>>> type(y)

<type 'numpy.ndarray'>
```

Internally data can be ordered either in C order or in Fortran order. While the arrays automatically created by f2py's modules are all in Fortran order that is not the default for numpy.array(). It is a good habit to get into to always add the optional argument order='Fortran' to override this default.

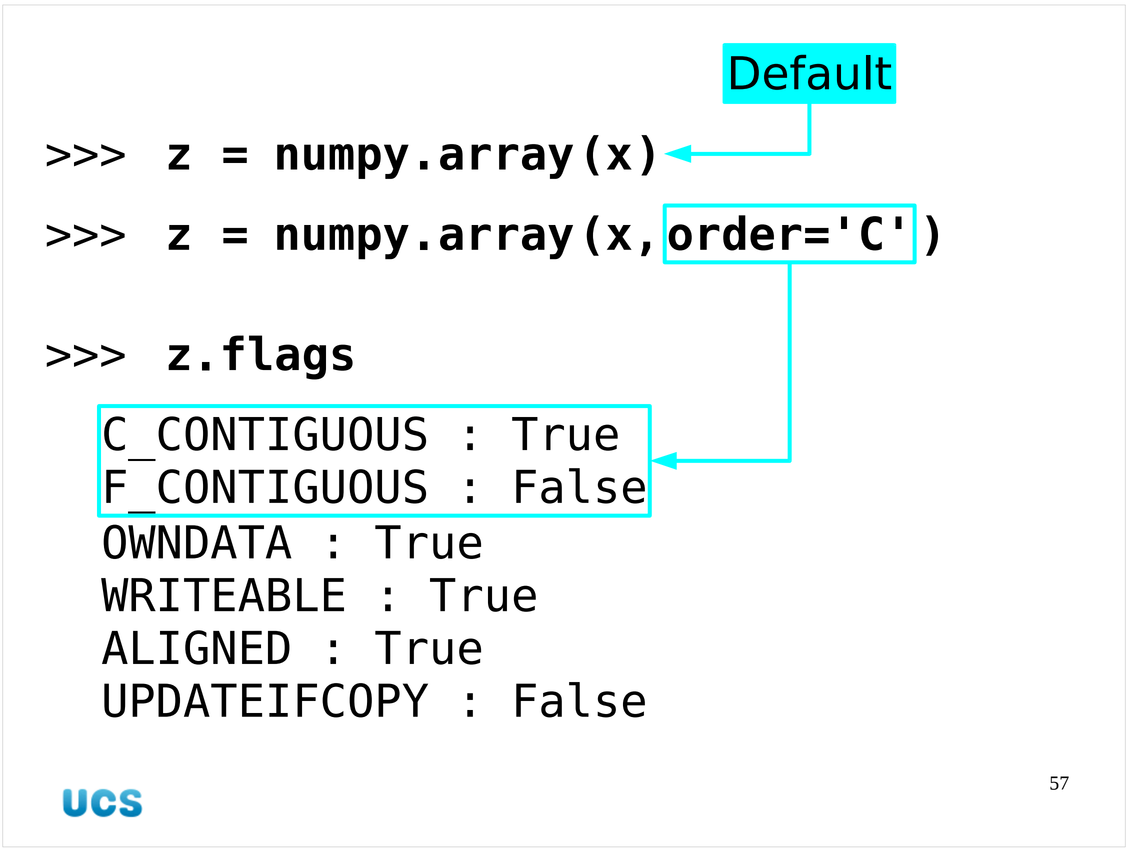


```
>>> y = numpy.array(x,order='Fortran')

>>> y.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

Each NumPy array comes with some flags expressing various of its properties. For the time being we are only interested in two of them.

C_CONTIGUOUS says whether or the data is internally laid out in the C order suitable for passing to a C function.

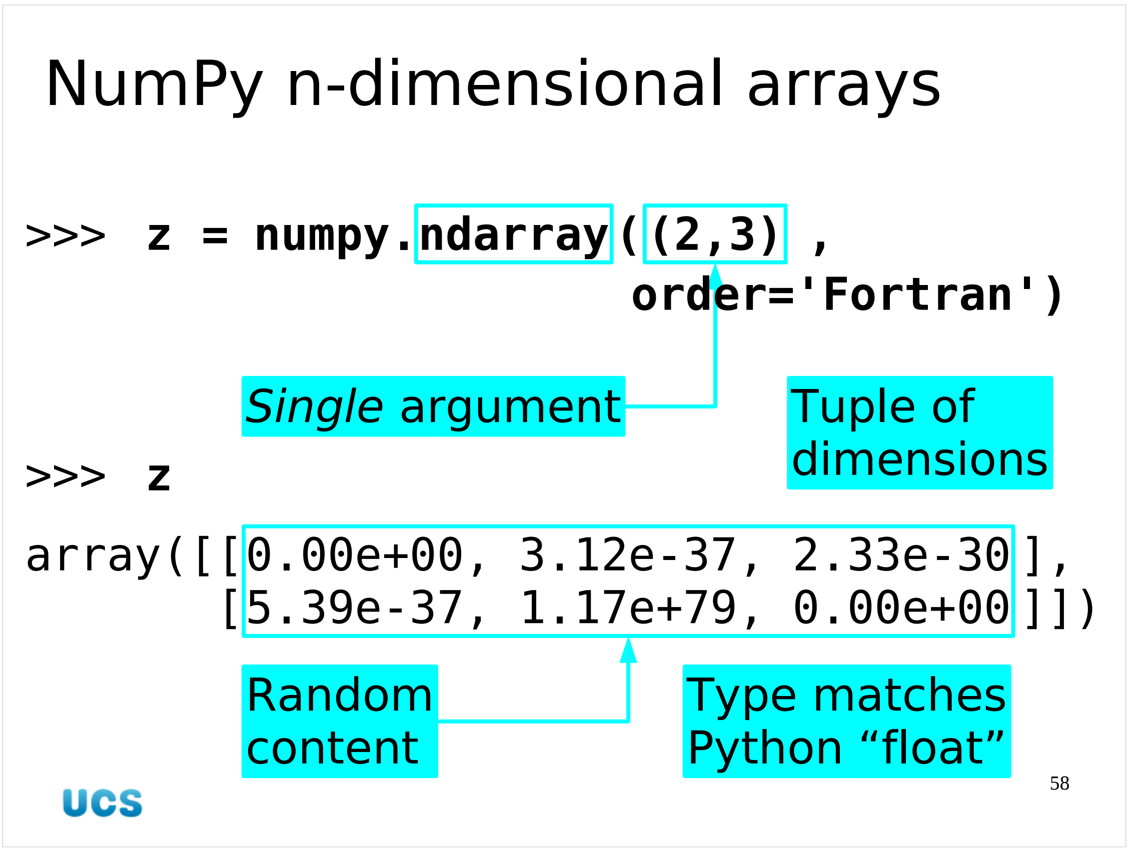
F_CONTIGUOUS says whether or the data is internally laid out in the Fortran order suitable for passing to a Fortran subroutine or function.



Default

```
>>> z = numpy.array(x)
>>> z = numpy.array(x,order='C')

>>> z.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

If we explicitly say that C ordering is to be used the two contiguity flags are switched round. Note that this is the *default*. Specifying the order is a a good habit to get into. There is no way to specify a different default.



NumPy n-dimensional arrays

```
>>>  z = numpy.ndarray((2,3) ,
                        order='Fortran')

>>>  z

array([[0.00e+00, 3.12e-37, 2.33e-30],
       [5.39e-37, 1.17e+79, 0.00e+00]])
```

Single argument

Tuple of dimensions

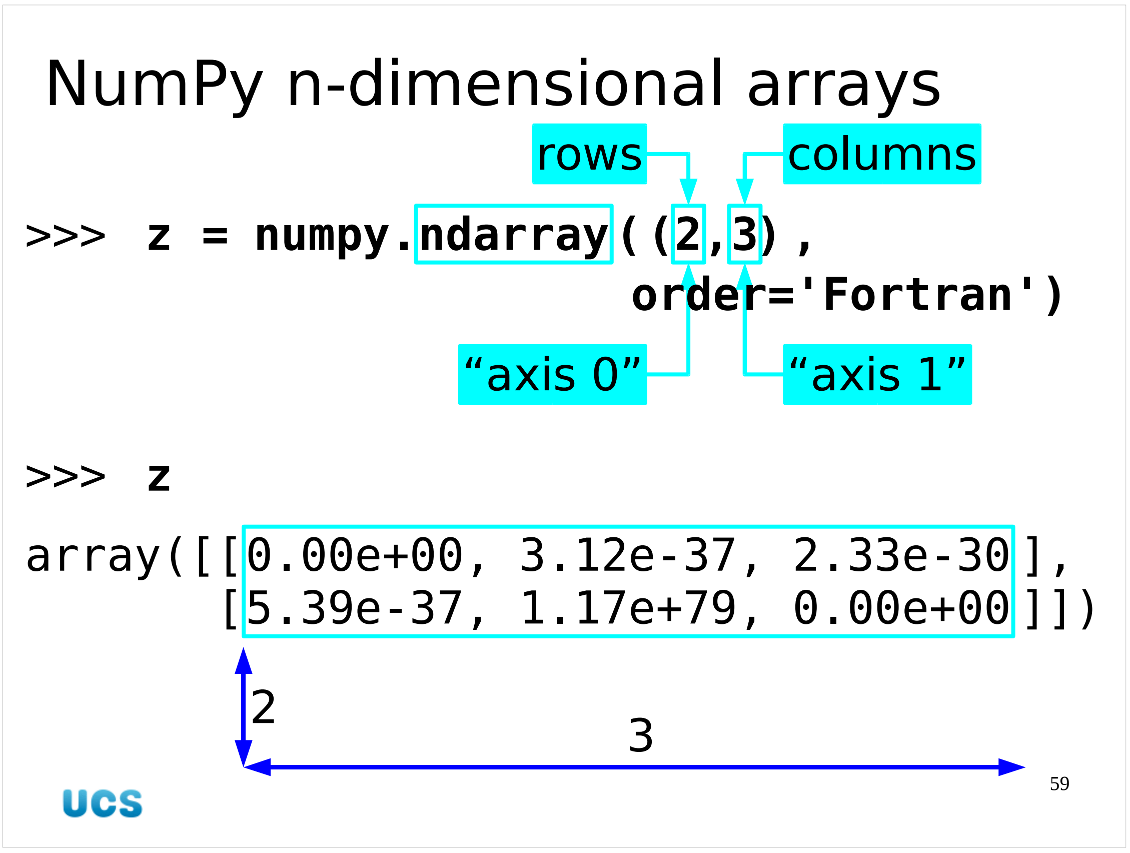
Random content

Type matches Python “float”

There is a constructor for NumPy arrays called numpy.ndarray().

It's first argument is a tuple of dimensions. Recall that a “single” is best written “(2,)”. It can also be passed the ordering argument.

Note that in true Fortran or C style (and unlike Python) it does not initialise its data so you get random muck in your object.



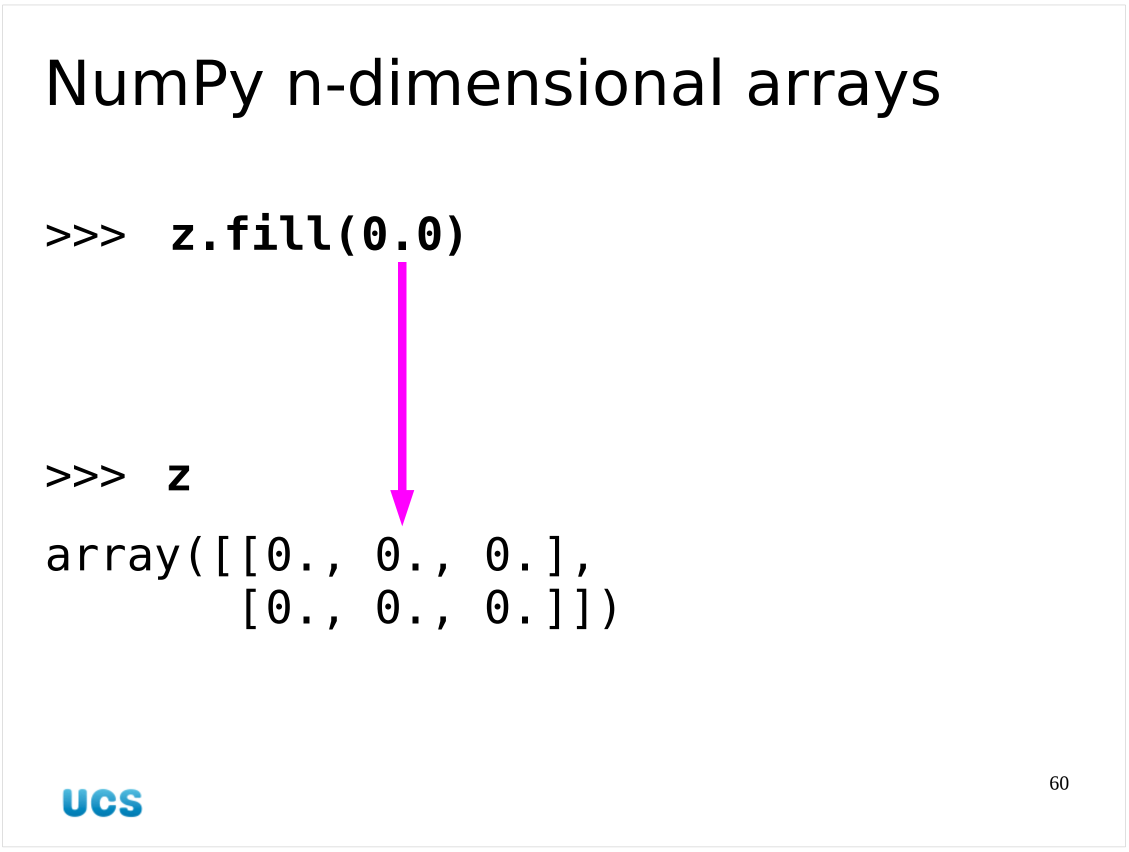
NumPy n-dimensional arrays

```
>>> z = numpy.ndarray((2,3),
                       order='Fortran')

>>> z

array([[0.00e+00, 3.12e-37, 2.33e-30],
       [5.39e-37, 1.17e+79, 0.00e+00]])
```

rows
columns
“axis 0”
“axis 1”
2
3

We will take the excuse of this tuple to look at the dimensions of a NumPy array.

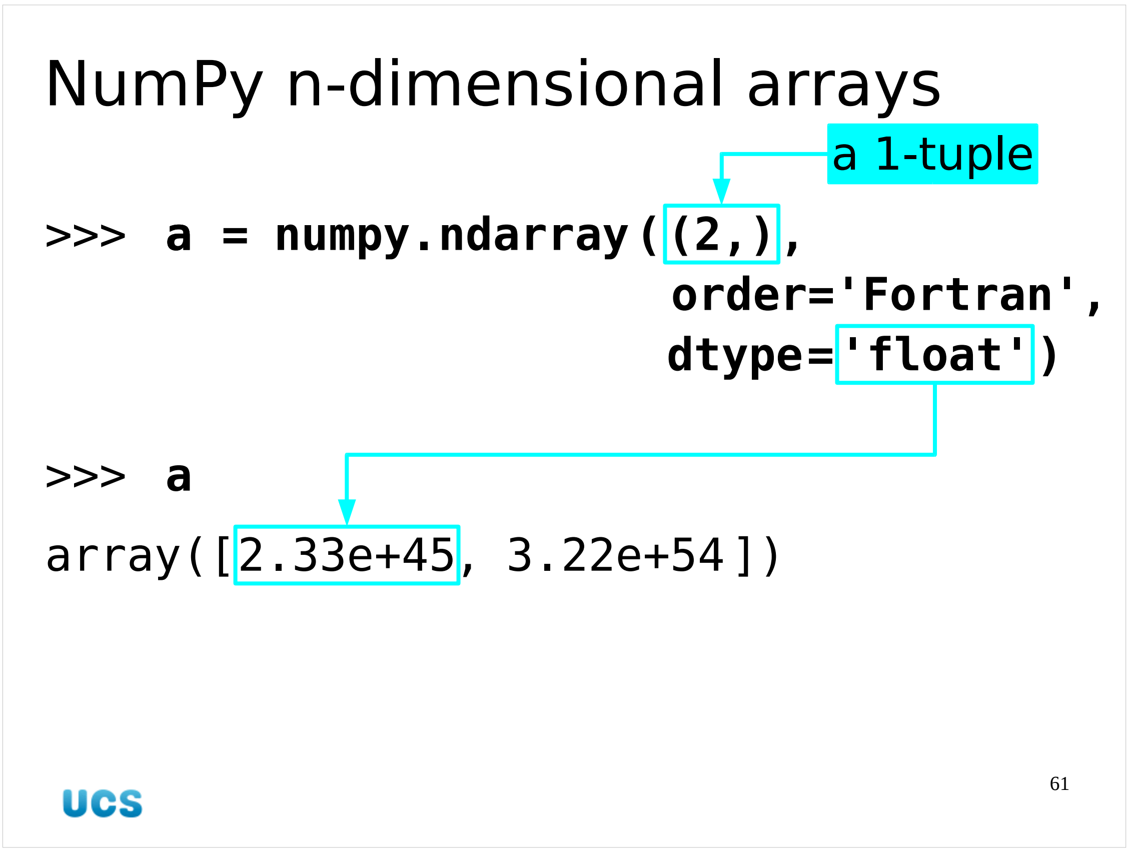
In the two-dimensional case the first number is the number of rows and the second the number of columns. Because we have to be able to generalise to an arbitrary number of dimensions NumPy avoids these two words and instead calls them “axis 0” and “axis 1” of the array. Axis 0 corresponds to the first index (remember that Python counts from zero!). Axis 1 corresponds to the second.



NumPy n-dimensional arrays

```
>>> z.fill(0.0)

>>> z

array([[0., 0., 0.],
       [0., 0., 0.]])
```

If you want your data initialised then the “fill()” method might be useful. This is passed one value of the same base type as the array and sets every element to that value.



NumPy n-dimensional arrays

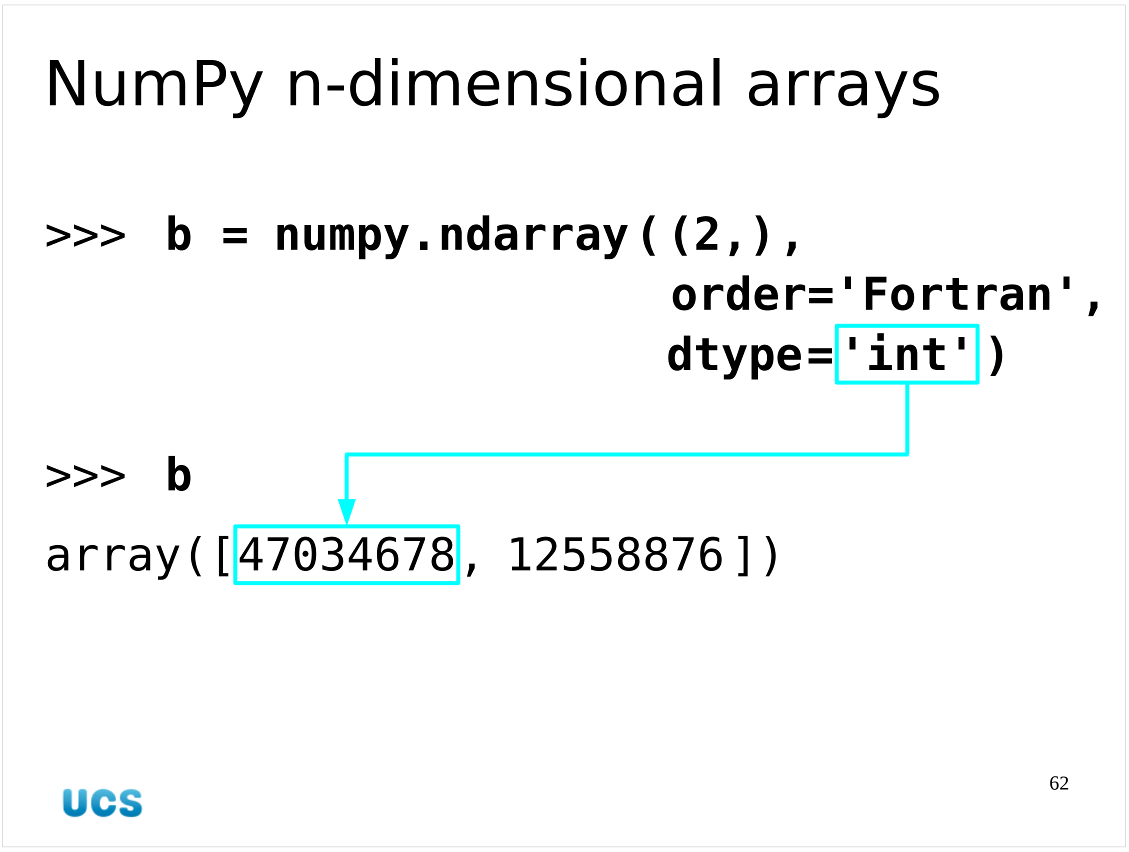
a 1-tuple

```
>>>  a = numpy.ndarray((2,),
                       order='Fortran',
                       dtype='float')

>>>  a
array([2.33e+45, 3.22e+54])
```

But how do I specify that I want an array of floating point numbers (the default) as opposed to integers or complex numbers (also available)?

There is another optional argument, dtype (“data type”), which specifies the type. This can take a number of values but we'll restrict our attention to the most useful.

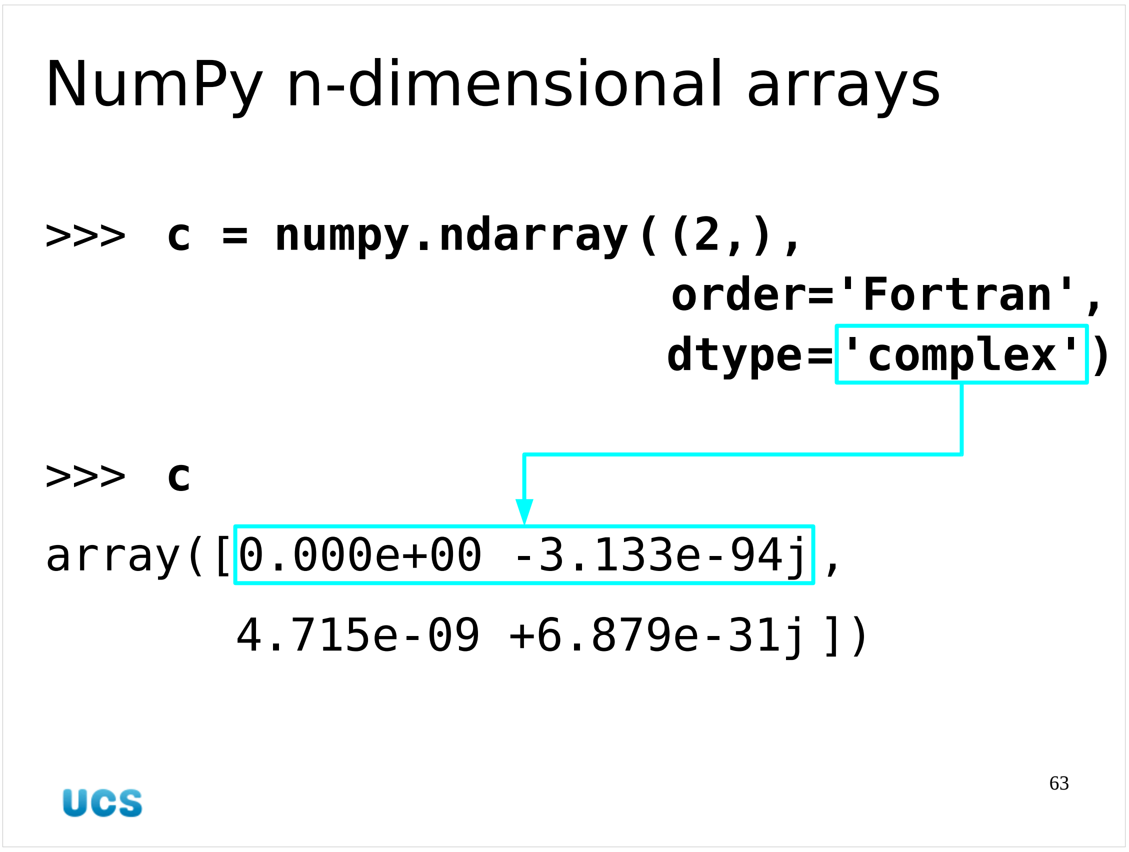
dtype='float' gets Fortran double precision floating point numbers.



NumPy n-dimensional arrays

```
>>> b = numpy.ndarray((2,),
                       order='Fortran',
                      dtype='int')

>>> b
array([47034678, 12558876])
```

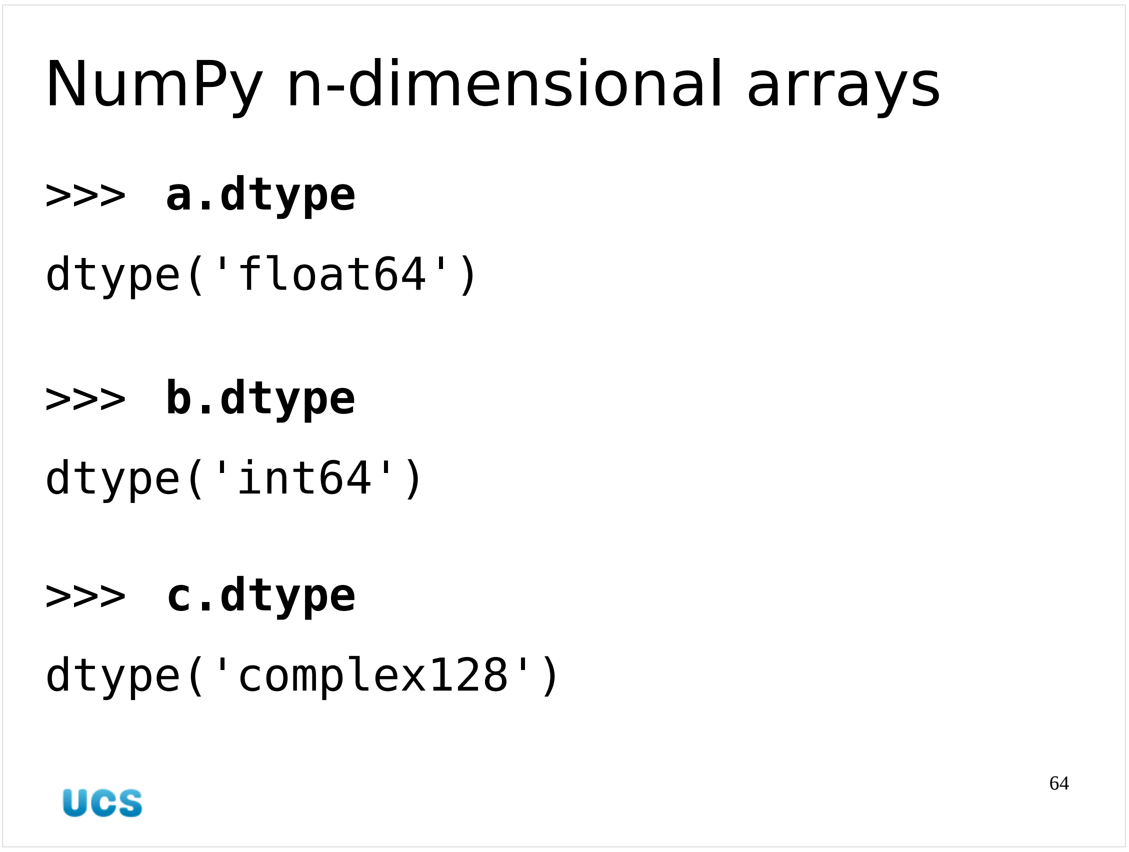
dtype='int' gets integers.



NumPy n-dimensional arrays

```
>>>  c = numpy.ndarray((2,),
                              order='Fortran',
                              dtype='complex')

>>>  c

array([0.000e+00 -3.133e-94j ,
        4.715e-09 +6.879e-31j ])
```

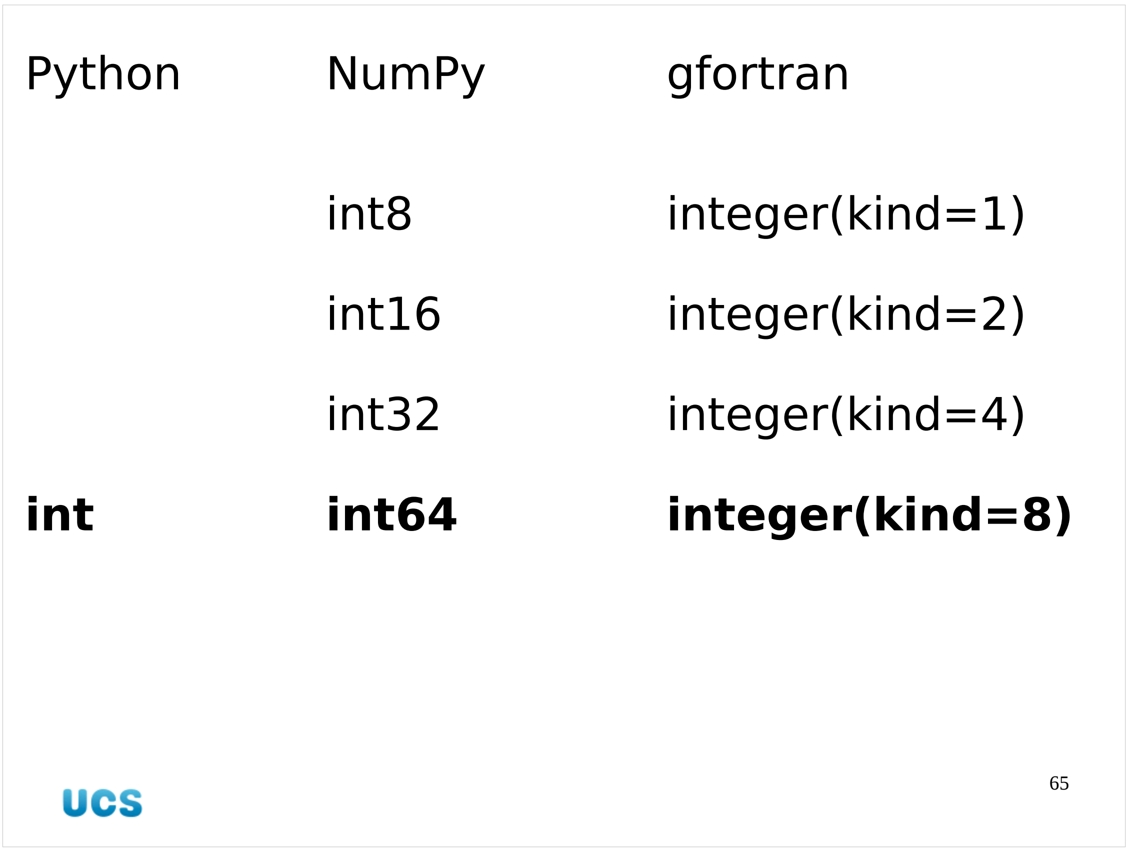
dtype='complex' gets double precision complex numbers.



NumPy n-dimensional arrays

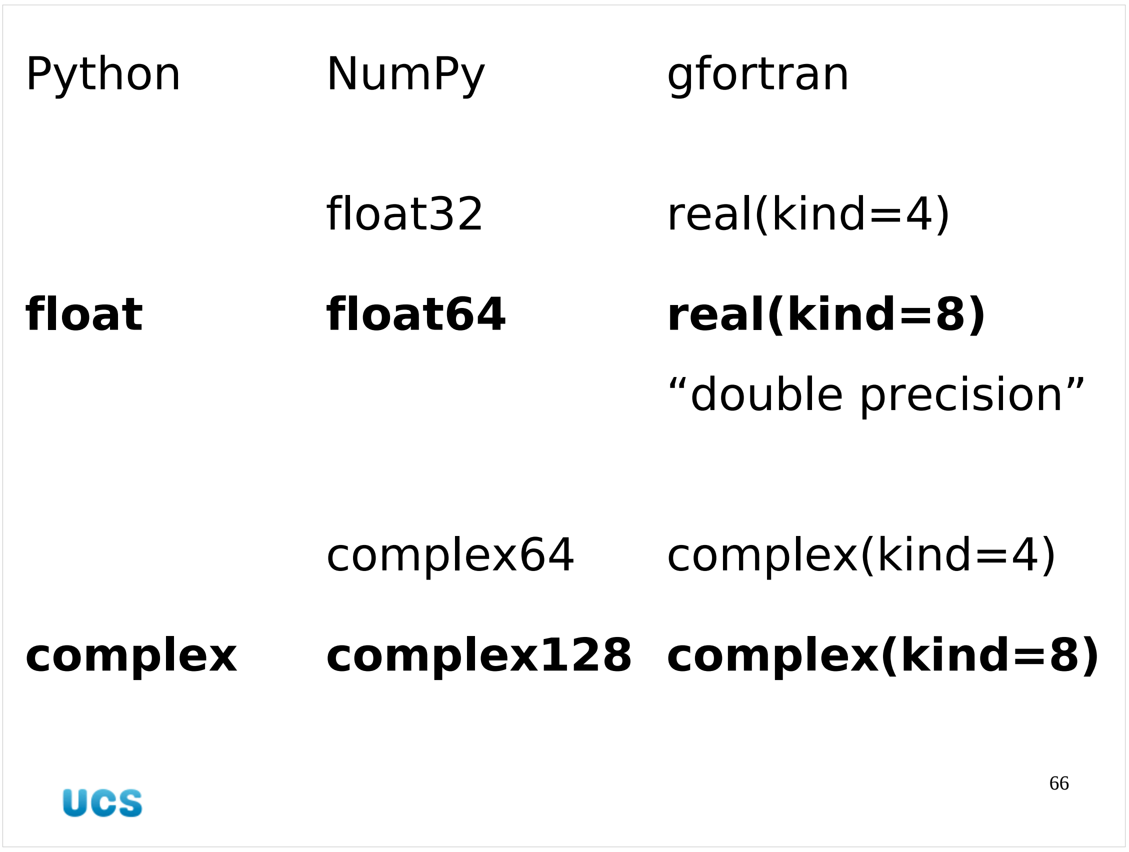
```
>>> a.dtype
dtype('float64')

>>> b.dtype
dtype('int64')

>>> c.dtype
dtype('complex128')
```

We can ask for the data back again with the dtype attribute. This lets us see the detailed type information corresponding to the simple aliases we used to create the arrays.



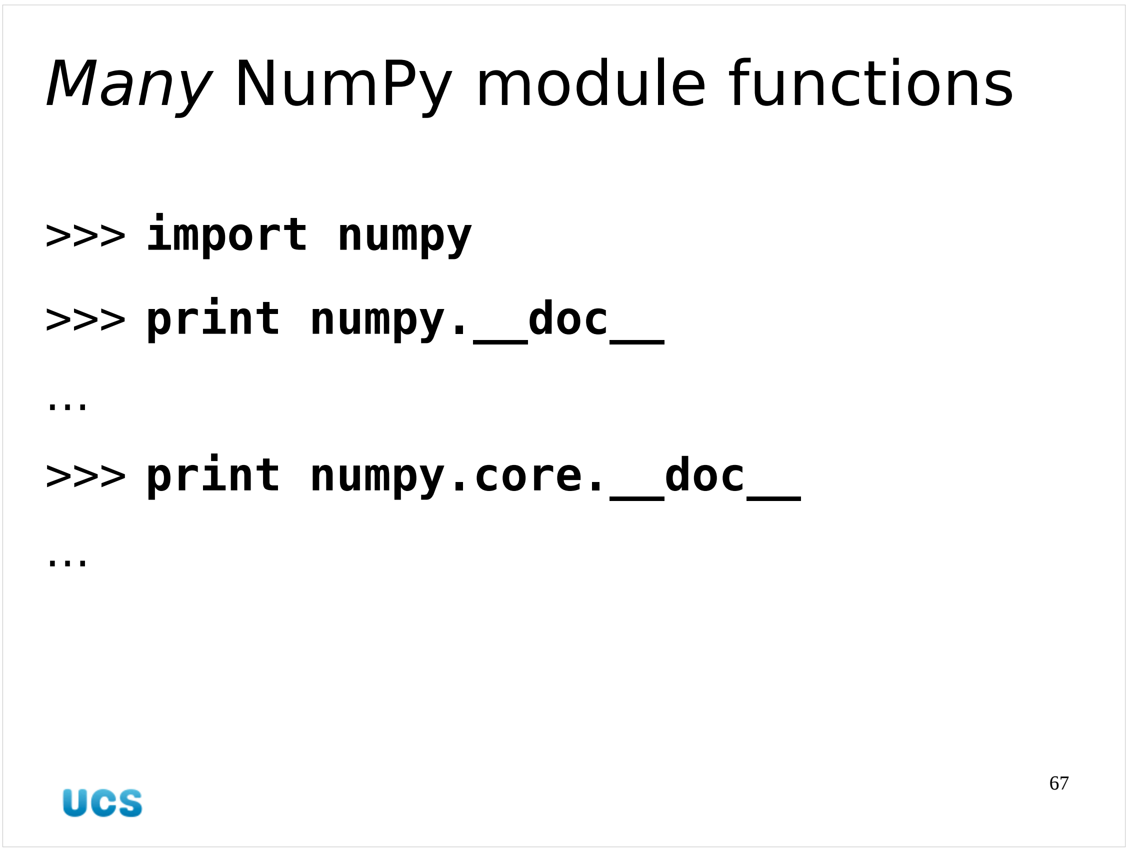
Python	NumPy	gfortran
	int8	integer(kind=1)
	int16	integer(kind=2)
	int32	integer(kind=4)
int	**int64**	**integer(kind=8)**

If you really want to see the grubby details here they are for integers…



Python	NumPy	gfortran
	float32	real(kind=4)
float	**float64**	**real(kind=8)**
		“double precision”
	complex64	complex(kind=4)
complex	**complex128**	**complex(kind=8)**

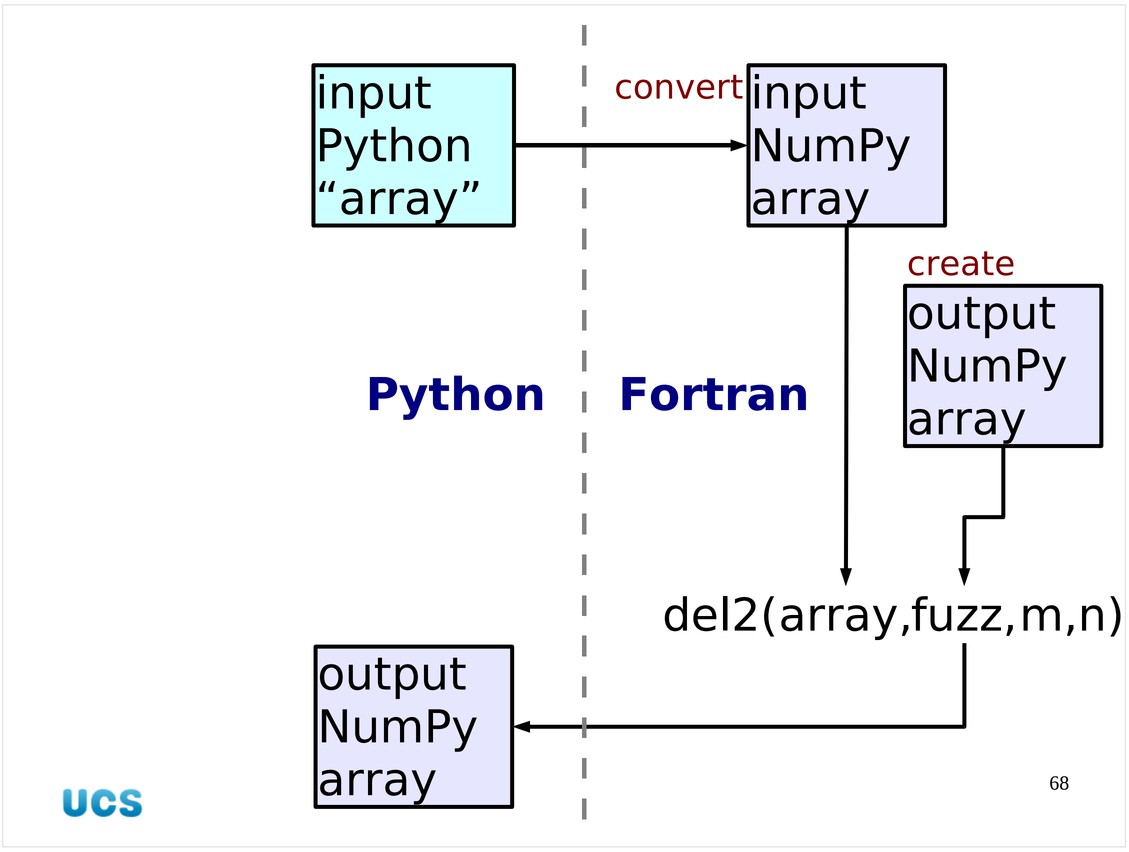
…and for floating point numbers.



Many NumPy module functions

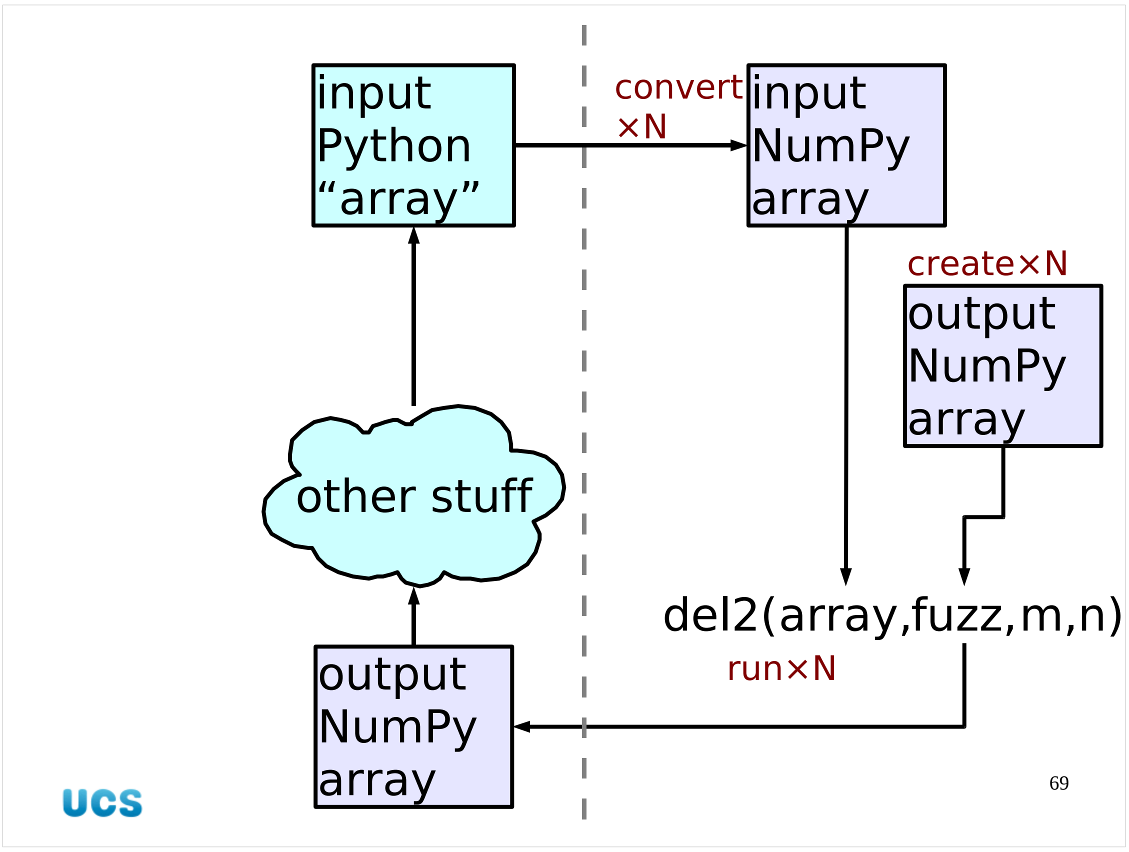
```
>>> import numpy
>>> print numpy.__doc__
…
>>> print numpy.core.__doc__
…
```

The NumPy module has *a lot* of functionality. Without touching the Fortran integration it could easily consume a course itself.

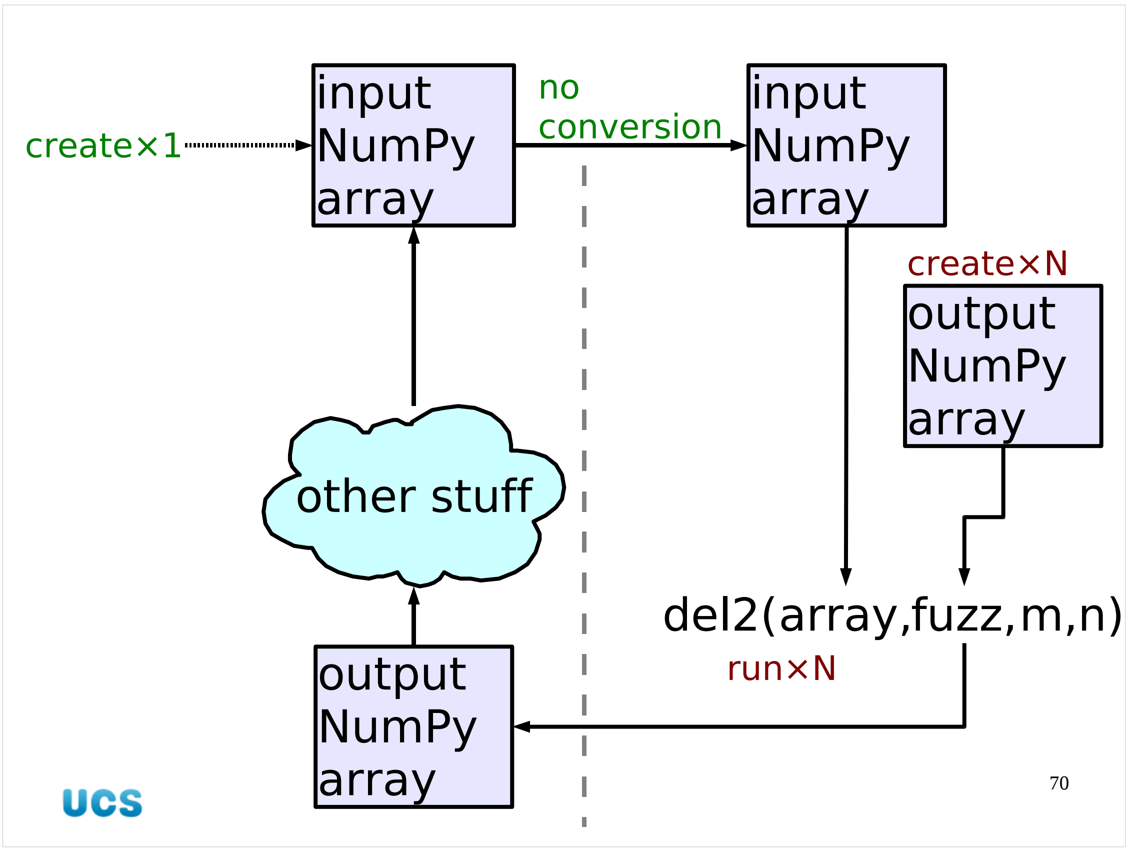


Finally, we will consider a small efficiency improvement that we can make. Our greatest gain, of course, is that the core numerical routine is written in Fortran and compiled to machine code rather than being written in Python and interpreted.

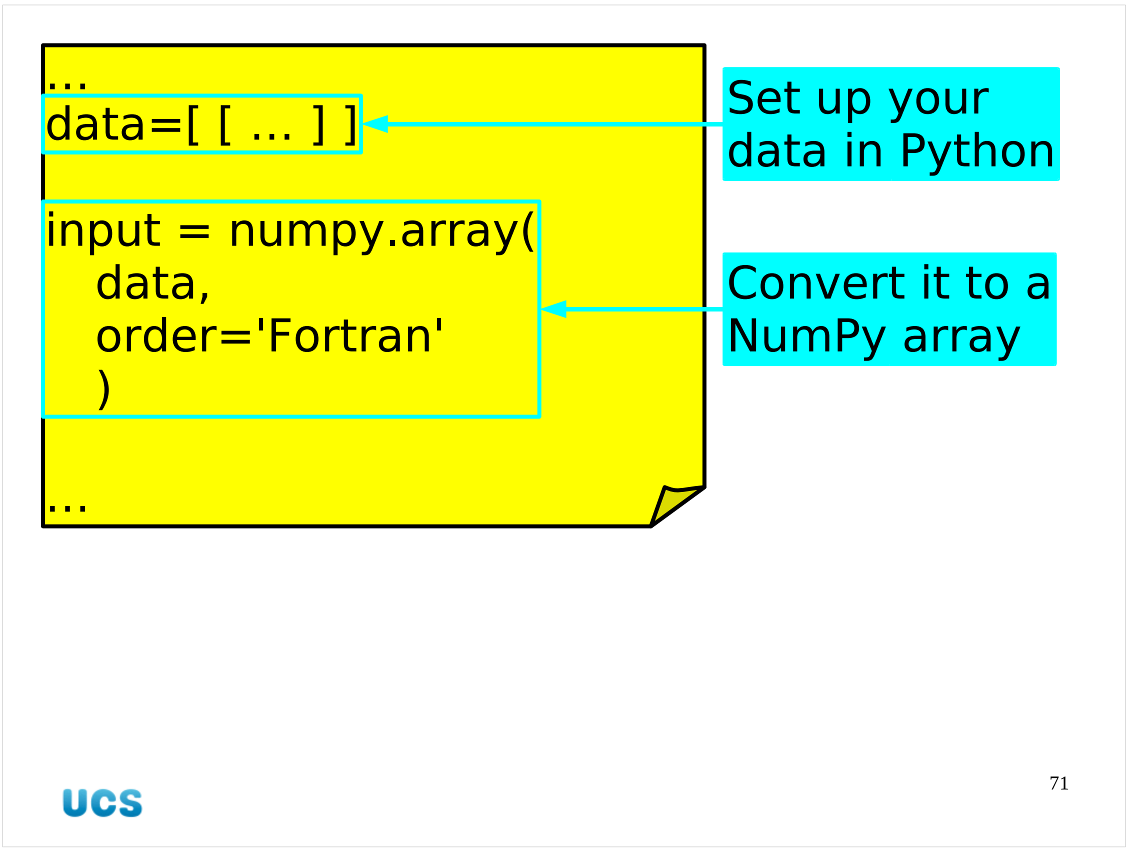
The input Python list of lists gets automatically converted to a NumPy array. This takes time. Similarly the output NumPy array has to be created to be passed to the Fortran subroutine.



These are just one-off conversions and creations. However, they can mount up in any looping environment where the f2py-created module function is used repeatedly.



First, we can eliminate the conversion of a Python list of lists to a NumPy array if we can reuse the same array in each iteration of the loop. We still have to set its values each loop but we are spared the actual creation of the object.

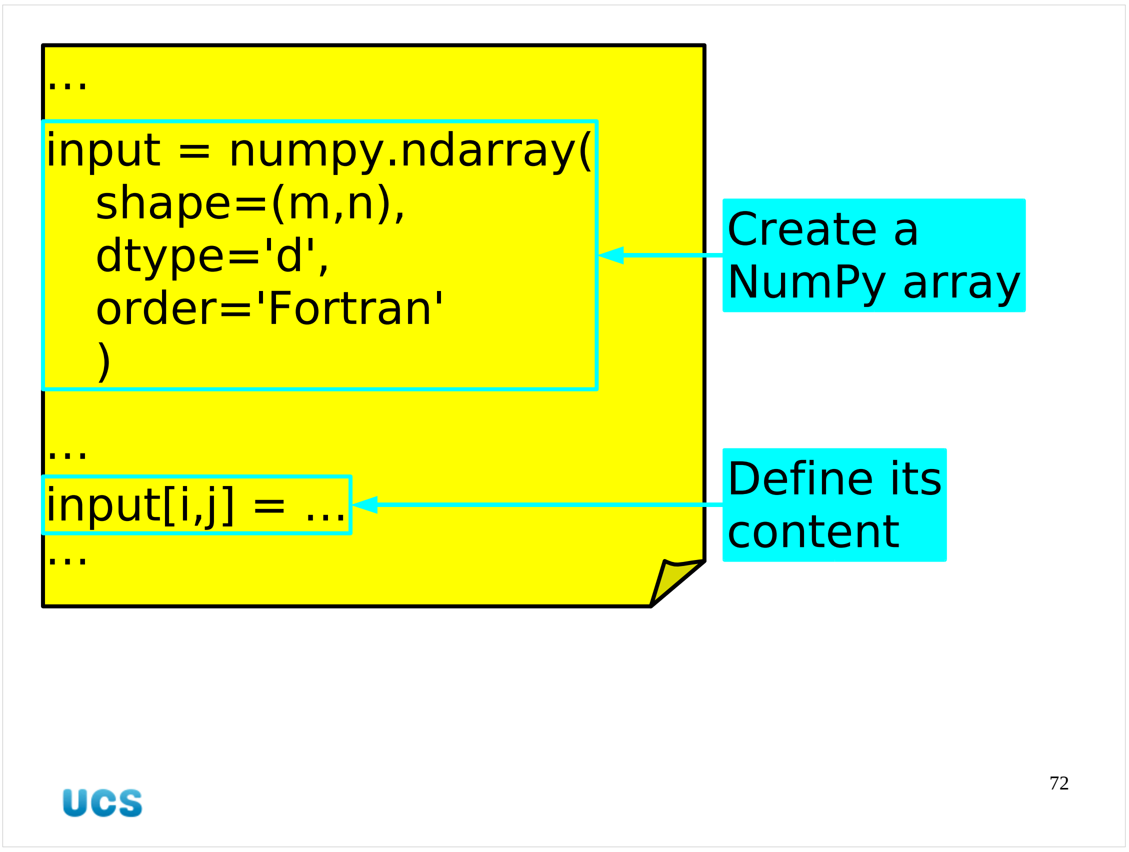


```
…
data=[ [ … ] ]

input = numpy.array(
   data,
   order='Fortran'
   )

…
```

Set up your data in Python

Convert it to a NumPy array

We can create a Python list of lists outside the loop and convert it to a NumPy array outside the loop. Any changes to the values we want to make inside the loop have to be made to the NumPy array. So this is a rather silly way to create a NumPy array.



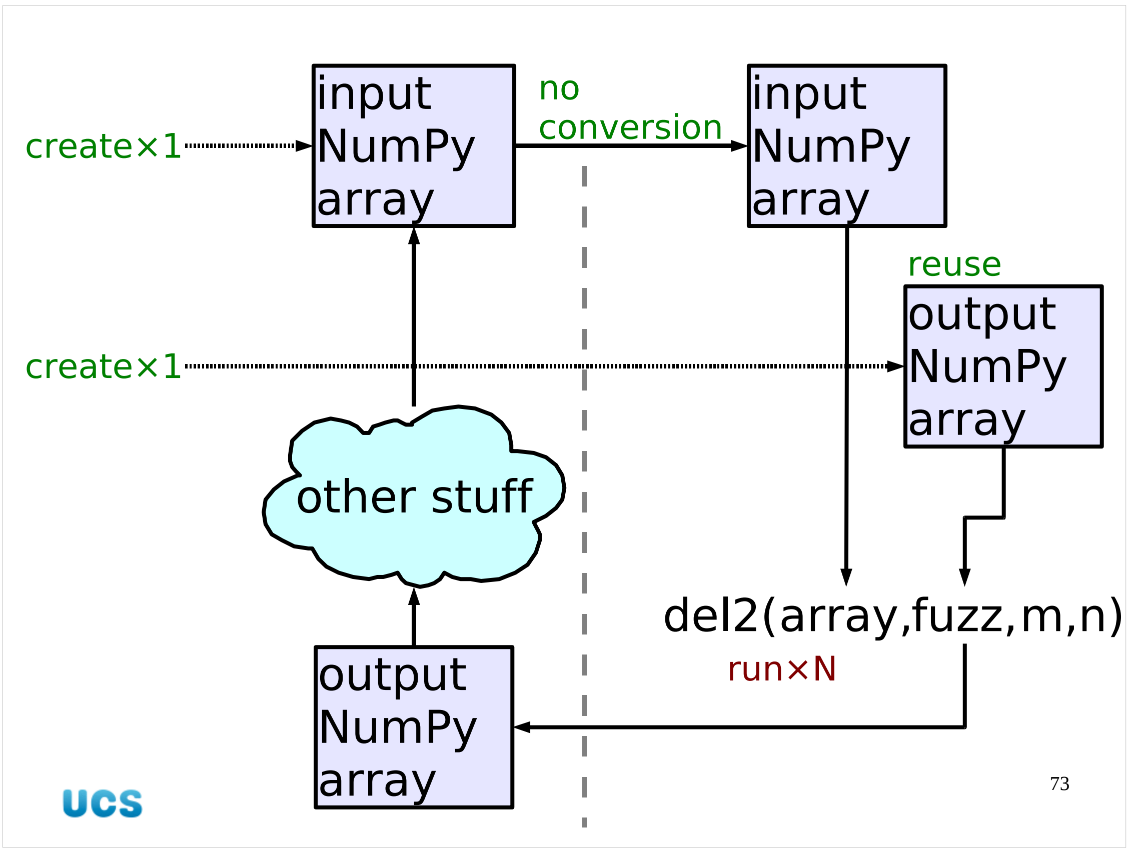
```
…
input = numpy.ndarray(
   shape=(m,n),
   dtype='d',
   order='Fortran'
   )
…
input[i,j] = …
…
```

Create a NumPy array

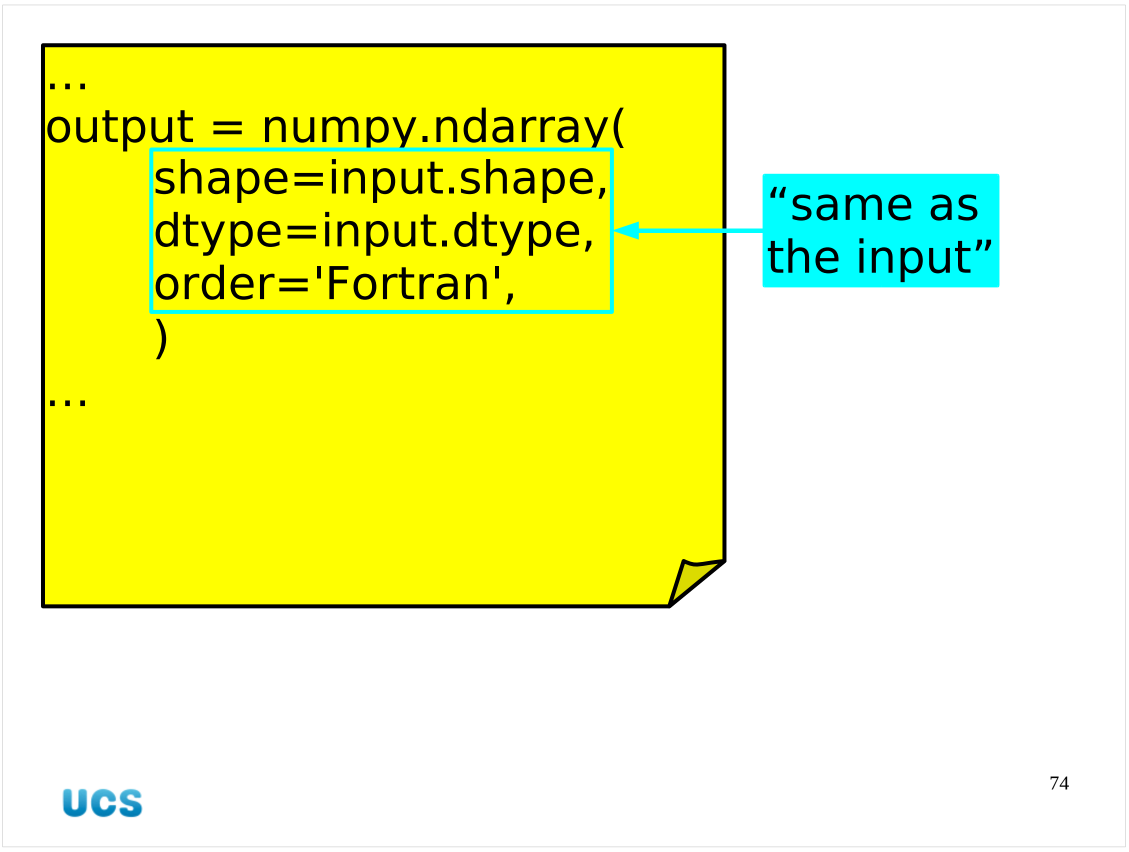
Define its content

Better is to create the NumPy array directly and then set its values.

Recall that the numpy.array() function converts Python lists into NumPy arrays. The numpy.ndarry() function creates NumPy arrays directly. (In object-oriented speak it's a “factory method”.)



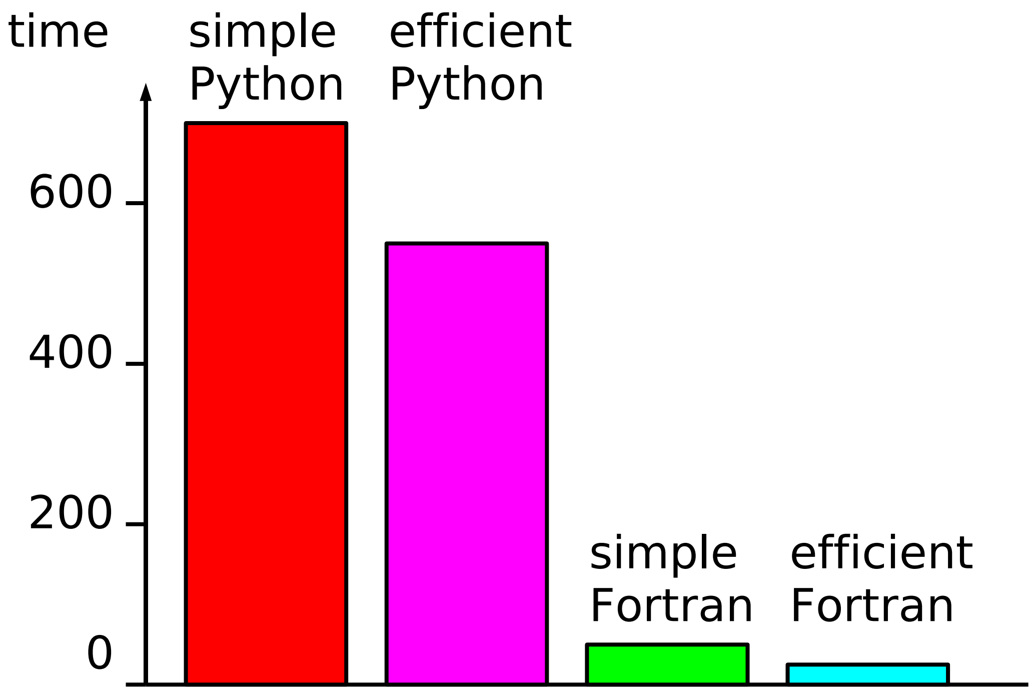
Similarly, if we can provide the embedded Fortran subroutine with the same NumPy array each time and just have the Fortran subroutine overwrite its previous values we are saved the effort of creating it each time too.



```
…
output = numpy.ndarray(
        shape=input.shape,
        dtype=input.dtype,
        order='Fortran',
        )
…
```

“same as the input”

So as well as creating the input NumPy array, we create the output one too. Note how we can make sure it matches the input array exactly.



So is it worth it?

Here are some timing tests for a 3,000×2,000 grid, each performed a hundred times. The absolute values are meaningless but the ratios are revealing.

Again, the major saving is in converting the core subroutine from Python to Fortran. We do get further savings from not recreating objects unnecessarily, in both the pure Python and mixed Python/Fortran worlds.

But converting from Python to Fortran gains us a factor of 10 speed improvement. Converting to efficient Fortran gains us a factor of 20.

Not bad for an afternoon's work.

f2py supported compilers

The complete set of Fortran compilers known to your instance of `f2py` can be ascertained with the command "`f2py -c --help-fcompiler`".

The complete set for version 2.4422 of `f2py` is given here. Compilers are identified by key with the options "`f2py -c --fcompiler=`*`key`* …".

Key	**Description of compiler**
`absoft`	Absoft Corporation Fortran Compiler
`compaq`	Compaq Fortran Compiler
`g95`	G95 Fortran Compiler (`g95`)
`gnu95`	GNU Fortran 95 compiler (`gfortran`, the successor to `g95`)
`gnu`	GNU Fortran 77 compiler (`g77`)
`hpux`	HP Fortran 90 Compiler
`ibm`	IBM XL Fortran Compiler
`intel`	Intel Fortran Compiler for 32-bit apps
`intele`	Intel Fortran Compiler for Itanium applications
`intelem`	Intel Fortran Compiler for EM64T-based applications
`intelev`	Intel Visual Fortran Compiler for Itanium applications
`intelv`	Intel Visual Fortran Compiler for 32-bit applications
`lahey`	Lahey/Fujitsu Fortran 95 Compiler
`mips`	MIPSpro Fortran Compiler
`nag`	NAGWare Fortran 95 Compiler
`pg`	Portland Group Fortran Compiler
`sun`	Sun or Forte Fortran 95 Compiler
`vast`	Pacific-Sierra Research Fortran 90 Compiler