



Pablo Alcain
pabloalcain@gmail.com

Interfaz de C/fortran con
Python



Python

Muchísimas librerías

Sintaxis limpia

No hace falta compilar

Duck typing

Súper versátil

Python

Muchísimas librerías

Sintaxis limpia

No hace falta compilar

Duck typing

Súper versátil

```
# file: add_numbers.py
total = 100000000
for i in xrange(10):
    avg = 0.0
    for j in xrange(total):
        avg += j
    avg = avg/total
print "Average is
{0}".format(avg)
```

```
/* file: add_numbers.c */
#include <stdio.h>
int main(int argc, char **argv) {
    int i, j, total;
    double avg;
    total = 100000000;
    for (i = 0; i < 10; i++) {
        avg = 0;
        for (j = 0; j < total; j++) {
            avg += j;
        }
        avg = avg/total;
    }
    printf("Average is %f\n", avg);
}
```

Pero la performance, esa palabra

```
$ time python add_numbers.py  
Average is 4999999.5
```

```
real 0m8.047s  
user 0m7.884s  
sys 0m0.012s
```

```
$ time ./add_numbers.e  
Average is 4999999.500000
```

```
real 0m0.284s  
user 0m0.283s  
sys 0m0.001s
```

Pero la performance, esa palabra

```
$ time python add_numbers.py  
Average is 4999999.5
```

```
real 0m8.047s  
user 0m7.884s  
sys 0m0.012s
```

```
$ time ./add_numbers.e  
Average is 4999999.500000
```

```
real 0m0.284s  
user 0m0.283s  
sys 0m0.001s
```

28x: una simulación de 1 hora tarda 1 día!

¿Qué cuenta Numpy?

```
# file: add_numbers_fast.py
from numpy import mean, arange
total = 100000000
a = arange(total)
for i in xrange(10):
    avg = mean(a)
print "Average is {0}".format(avg)
```

¿Qué cuenta Numpy?

```
# file: add_numbers_fast.py
from numpy import mean, arange
total = 100000000
a = arange(total)
for i in xrange(10):
    avg = mean(a)
print "Average is {0}".format(avg)
```

```
$ time python add_numbers_fast.py
Average is 49999999.5
```

```
real 0m0.266s
user 0m0.189s
sys 0m0.075s
```

Comparable con C

¿Qué cuenta Numpy?

```
# file: add_numbers_fast.py
from numpy import mean, arange
total = 100000000
a = arange(total)
for i in xrange(10):
    avg = mean(a)
print "Average is {0}".format(avg)
```

```
$ time python add_numbers_fast.py
Average is 49999999.5
```

```
real 0m0.266s
user 0m0.189s
sys 0m0.075s
```

Comparable con C

(Pero no es muy versátil, ¡sólo puedo usar vectores!)

Solución manual

Si cualquier lenguaje compilado se puede linkear con cualquier otro, tiene que haber alguna forma de linkear Python con C.

1. Escribir el código en Python
2. Escribir la parte que consume tiempo en C
3. Escribir una API C/Python

¿Esto funciona? ¿La gente lo hace?

Solución manual

Si cualquier lenguaje compilado se puede linkear con cualquier otro, tiene que haber alguna forma de linkear Python con C.

1. Escribir el código en Python
2. Escribir la parte que consume tiempo en C
3. Escribir una API C/Python

¿Esto funciona? ¿La gente lo hace?

NUMPY

Herramientas

Construir el módulo de Python en C

(Si Python se hizo en C, cualquier cosa de python se puede hacer en C)

Ctypes

(Llamar a funciones de C definiendo los tipos en Python)

Cython

(Autogenerador de código de C desde Python)

F2PY

(Sólo para FORTRAN, fácil de usar, algunos problemas con memoria)

Boost, SWIG, ...

(Boost sirve mucho para C++, SWIG es para muchísimos lenguajes)

Herramientas

Construir el módulo de Python en C

(Si Python se hizo en C, cualquier cosa de python se puede hacer en C)

Ctypes

(Llamar a funciones de C definiendo los tipos en Python)

Cython

(Autogenerador de código de C desde Python)

F2PY

(Sólo para FORTRAN, fácil de usar, algunos problemas con memoria)

Boost, SWIG, ...

(Boost sirve mucho para C++, SWIG es para muchísimos lenguajes)

Librería

Partimos de una librería en C

```
/* file: add_two.c */

float add_float(float a, float b) {
    return a + b;
}

int add_int(int a, int b) {
    return a + b;
}

int add_float_ref(float *a,
                  float *b, float *c) {
    *c = *a + *b;
    return 0;
}

int add_int_ref(int *a, int *b, int *c) {
    *c = *a + *b;
    return 0;
}
```

```
/* file: arrays.c */

int add_int_array(int *a, int *b,
                  int *c, int n) {
    int i;
    for (i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
    return 0;
}

float dot_product(float *a,
                  float *b, int n) {
    float res;
    int i;
    res = 0;
    for (i = 0; i < n; i++) {
        res = res + a[i] * b[i];
    }
    return res;
}
```

Librería

```
$ gcc -c -fPIC arrays.c
$ gcc -c -fPIC add_two.c
$ gcc -shared arrays.o add_two.o -o libmymath.so
$ nm -n libmymath.so

. . .
000000000000000730 T add_float_array
0000000000000008a0 T dot_product
0000000000000008d0 T add_float
0000000000000008e0 T add_int
0000000000000008f0 T add_float_ref
000000000000000900 T add_int_ref
. . .
```

Ctypes

ctypes type	C type
<code>c_bool</code>	<code>_Bool</code>
<code>c_char</code>	<code>char</code>
<code>c_wchar</code>	<code>wchar_t</code>
<code>c_byte</code>	<code>char</code>
<code>c_ubyte</code>	<code>unsigned char</code>
<code>c_short</code>	<code>short</code>
<code>c_ushort</code>	<code>unsigned short</code>
<code>c_int</code>	<code>int</code>
<code>c_uint</code>	<code>unsigned int</code>
<code>c_long</code>	<code>long</code>
<code>c_ulong</code>	<code>unsigned long</code>
<code>c_longlong</code>	<code>__int64</code> or <code>long long</code>
<code>c_ulonglong</code>	<code>unsigned __int64</code> or <code>unsigned long long</code>
<code>c_float</code>	<code>float</code>
<code>c_double</code>	<code>double</code>
<code>c_longdouble</code>	<code>long double</code>
<code>c_char_p</code>	<code>char *</code> (NUL terminated)
<code>c_wchar_p</code>	<code>wchar_t *</code> (NUL terminated)
<code>c_void_p</code>	<code>void *</code>

- Tipos de C
- Dynamic Loader

Llamando a una función de la librería

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> math.add_int(3, 4)
7
```


Ctypes

```
>>> math.add_float(3, 4)
```

Ctypes

```
>>> math.add_float(3, 4)  
0
```

Ctypes

```
>>> math.add_float(3, 4)
0
>>> math.add_float(3.0, 4.0)
```

Ctypes

```
>>> math.add_float(3, 4)
0
>>> math.add_float(3.0, 4.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <type
'exceptions.TypeError'>: Don't know how to convert
parameter 1
```

Ctypes

```
>>> math.add_float(3, 4)
0
>>> math.add_float(3.0, 4.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <type
'exceptions.TypeError'>: Don't know how to convert
parameter 1
>>> math.add_float(C.c_float(3.0), C.c_float(4.0))
```

Ctypes

```
>>> math.add_float(3, 4)
0
>>> math.add_float(3.0, 4.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <type
'exceptions.TypeError'>: Don't know how to convert
parameter 1
>>> math.add_float(C.c_float(3.0), C.c_float(4.0))
2
```

Ctypes: Ahora funciona

```
>>> math.add_float.restype = C.c_float  
>>> math.add_float(C.c_float(3.0), C.c_float(4.0))  
7.0
```

Ctypes: La forma más limpia

```
>>> math.add_float.restype = C.c_float  
>>> math.add_float.argtypes = [C.c_float, C.c_float]  
>>> math.add_float(3, 4)  
7.0
```


Ctypes: Por referencia a int

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> tres = C.c_int(3)
>>> cuatro = C.c_int(4)
>>> res = C.c_int()
>>> math.add_int_ref(C.byref(tres),
                    C.byref(cuatro),
                    C.byref(res))

0
>>> res.value
7
```

Ctypes: Por referencia a float

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> tres = C.c_float(3)
>>> cuatro = C.c_float(4)
>>> res = C.c_float()
>>> math.add_float_ref(C.byref(tres),
                        C.byref(cuatro),
                        C.byref(res))

0
>>> res.value
7.0
```

¿Por qué ahora funciona?

Ctypes: Arrays

A priori no debería ser muy difícil; es una llamada por referencia, apuntando al primer elemento

El problema es, como siempre, el memory management.

Manejar siempre memoria en Python

¿Cómo alocamos un array en Python?

Ctypes: Arrays

A priori no debería ser muy difícil; es una llamada por referencia, apuntando al primer elemento

El problema es, como siempre, el memory management.

Manejar siempre memoria en Python

¿Cómo alocamos un array en Python?

Ctypes: Arrays

```
>>> import ctypes as C
>>> math = C.CDLL('./libmymath.so')
>>> in1 = (C.c_int * 3) (1, 2, -5)
>>> in2 = (C.c_int * 3) (-1, 3, 3)
>>> math.add_int_array(C.byref(in1),
                        C.byref(in2),
                        C.byref(out))

0
>>> out[0], out[1], out[2]
(0, 5, -2)
```

Ctypes: Numpy Arrays

```
>>> import ctypes as C
>>> import numpy as np
>>> intp = C.POINTER(C.c_int)
>>> math = C.CDLL('./libmymath.so')
>>> in1 = np.array([1, 2, -5], dtype=C.c_int)
>>> in2 = np.array([-1, 3, 3], dtype=C.c_int)
>>> out = np.zeros(3, dtype=np.float16)
>>> math.add_int_array(in1.ctypes.data_as(flp),
                        in2.ctypes.data_as(flp),
                        out.ctypes.data_as(flp),
                        C.c_int(3))

0
>>> out
array([ 0,  5, -2], dtype=int32)
```

Ctypes: Structures

```
/* file: rectangle.c */  
struct _rect {  
    float height, width;  
};  
  
typedef struct _rect Rectangle;  
  
float area(Rectangle rect) {  
    return rect.height * rect.width;  
}
```

¿Cómo aprovechamos esta estructura como un objeto en python?

```
$ gcc -fPIC -c rectangle.c  
$ gcc -shared rectangle.o -o libgeometry.so
```

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("width", C.c_float),
                ("height", C.c_float)]

    def init(width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```


Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("width", C.c_float),
                ("height", C.c_float)]

    def init(width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("width", C.c_float),
                ("height", C.c_float)]

    def init(width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Hereda de C struct

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("width", C.c_float),
                ("height", C.c_float)]

    def init(width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Hereda de C struct

El mismo memory layout

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("width", C.c_float),
                ("height", C.c_float)]

    def init(width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Hereda de C struct

El mismo memory layout

Constructor

Ctypes: Structures

```
# file: geometry.py
import ctypes as C
CLIB = C.CDLL('./libgeometry.so')
CLIB.area.argtypes = [C.Structure]
CLIB.area.restype = C.c_float

class Rectangle(C.Structure):
    _fields_ = [("width", C.c_float),
                ("height", C.c_float)]

    def init(width, height):
        self.width = width
        self.height = height

    def area(self):
        return CLIB.area(self)
```

Llamar a las funciones

Hereda de C struct

El mismo memory layout

Constructor

Wrapper a C

Ctypes: Structures

```
>>> import geometry
>>> r = geometry.Rectangle(2, 3)
>>> r.area()
6.0
>>> r.width=10
>>> r.area()
30.0
```

La implementación en C está completamente encapsulada

Parte del paquete de numpy

Gran parte de numpy está portado con f2py

Genera una librería en FORTRAN directamente compatible con Python

Crea documentación automática bastante buena

Código creado por máquina, pero mucho mejor que Cython

Es un *transpiler*, de FORTRAN a FORTRAN

```
! file: fib1.f90
subroutine fib(a,n)
  !
  !   calculate first n fibonacci numbers
  !
  integer n
  real*8 a(n)
  do i=1,n
    if (i.eq.1) then
      a(i) = 0.0d0
    elseif (i.eq.2) then
      a(i) = 1.0d0
    else
      a(i) = a(i-1) + a(i-2)
    endif
  enddo
end subroutine fib
```


F2PY: La forma fácil

```
$ f2py -c fib1.f90 -m fib1
$ python
>>> import fib1
>>> import numpy
>>> a = numpy.zeros(8, dtype=numpy.float64)
>>> fib1(a, 8)
>>> print a
[  0.   1.   1.   2.   3.   5.   8.  13.]
```

F2PY: La forma fácil

```
$ f2py -c fib1.f90 -m fib1
$ python
>>> import fib1
>>> import numpy
>>> a = numpy.zeros(8, dtype=numpy.float64)
>>> fib1(a, 8)
>>> print a
[  0.   1.   1.   2.   3.   5.   8.  13.]
```

Ah, pero qué fácil!

F2PY: La forma piola

```
$ f2py fib1.f90 -m fib2 fib1.pyf
```

F2PY: La forma piola

```
$ f2py fib1.f90 -m fib2 fib2.pyf
```

```
!      -*- f90 -*-  
! Note: the context of this file is case sensitive.  
  
python module fib2 ! in  
    interface ! in :fib1  
        subroutine fib(a,n) ! in :fib1:fib1.f90  
            real*8 dimension(n) :: a  
            integer, optional, check(len(a)>=n), depend(a) :: n=len(a)  
        end subroutine fib  
    end interface  
end python module fib1  
  
! This file was auto-generated with f2py (version:2).  
! See http://cens.ioc.ee/projects/f2py2e/
```

F2PY: La forma piola

```
$ cat fib2.py
```

```
!      -*- f90 -*-  
! Note: the context of this file is case sensitive.  
  
python module fib2 ! in  
    interface ! in :fib1  
        subroutine fib(a,n) ! in :fib1:fib1.f90  
            real*8 dimension(n), intent(out), depend(n) :: a  
            integer intent(in) :: n  
        end subroutine fib  
    end interface  
end python module fib1  
  
! This file was auto-generated with f2py (version:2).  
! See http://cens.ioc.ee/projects/f2py2e/
```

F2PY: La forma piola

```
$ f2py -c fib1.f90 fib2.pyf
$ python
>>> import numpy
>>> import fib2
>>> a = fib2.fib(8)
>>> print a
[ 0.  1.  1.  2.  3.  5.  8. 13.]
```

F2PY: La forma piola y fácil

```
! file: fib3.f90
subroutine fib(a,n)
  !
  ! calculate first n fibonacci numbers
  !
  integer n
  real*8 a(n)
  !f2py intent(in) n
  !f2py intent(out) a
  !f2py depend(n) a
  do i=1,n
    if (i.eq.1) then
      a(i) = 0.0d0
    elseif (i.eq.2) then
      a(i) = 1.0d0
    else
      a(i) = a(i-1) + a(i-2)
    endif
  enddo
end subroutine fib
```

F2PY: La forma piola y fácil

```
! file: fib3.f90
subroutine fib(a,n)
  !
  !   calculate first n fibonacci numbers
  !
  integer n
  real*8 a(n)
  !f2py intent(in) n
  !f2py intent(out) a
  !f2py depend(n) a
  do i=1,n
    if (i.eq.1) then
      a(i) = 0.0d0
    elseif (i.eq.2) then
      a(i) = 1.0d0
    else
      a(i) = a(i-1) + a(i-2)
    endif
  enddo
end subroutine fib
```


F2PY: La forma piola y fácil

```
$ f2py -c fib3.f90 -m fib3
$ python
>>> import numpy
>>> import fib3
>>> a = fib3.fib(8)
>>> print a
[ 0.  1.  1.  2.  3.  5.  8. 13.]
```



Pablo Alcain
pabloalcain@gmail.com

Interfaz de C/fortran con
Python

