Comunicar python con c mediante ctypes.

Ctypes es un módulo que proporciona acceso directo a cualquier librería compartida (*so* en UNIX y *dll* en windows). Es parte de cpython desde la versión 2.5. Es preciso enfatizar el adjetivo *directo*, el módulo dota a python de los tipos necesarios y convierte cada biblioteca en un objeto manipulable desde el código. La consecuencia directa es una pequeña revolución en los lenguajes interpretados, hasta ahora era necesario escribir código para que la unión entre los lenguajes compilados e interpretados fuera suave, algo llamado *wrapper*. Este código ya no es necesario porque ctypes permite importar cualquier biblioteca compartida.

Para quien no haya asimilado aún el concepto he aquí un pequeño ejemplo del manual de ctypes

```
>>> from ctypes import *
>>> libc=cdll.LoadLibrary("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle b7ed0ce0 at -48408c14>
>>> libc.time
<_FuncPtr object at 0xb7d26b24>
>>> libc.time(None)
1178013320
```

Warning

El único cometido de ctypes es acceder a la biblioteca, no incluye per se ningún instrumento que ayude a no romper nada, si se hacen las cosas mal será común provocar violaciones de segmento. La mayoría serán producidos por no conocer la biblioteca con la que se está trabajando; por ejemplo, al llamar la función `time` sin argumento:

```
>>> libc.time()
Violación de segmento
```

Existen herramientas para impedir este tipo de errores que se analizarán en la sección de llamadas a funciones.

Al realizar una comunicación directa con C es necesario utilizar sus tipos. No hay una relación biunívoca entre los tipos de python y los de C, además, python es un lenguaje de tipos dinámics y C de tipos estáticos. Para utilizar ctypes se deben *declarar variables*, por lo menos las que las funciones de la biblioteca necesiten como argumentos.

Tipos básicos

No se tratarán todos los tipos básicos disponibles en ctypes, esto está perfectamente explicado en el manual. Bastará por ahora con unos ejemplos de tipos básicos escalares y lo que es más importante, cómo reservar memoria para arrays.

Por ejemplo, para declarar un entero:

```
>>> i=c_int(3)
>>> i
c_long(3)
```

¿Cómo es posible que al declarar un entero haya aparecido un entero de mayor precisión? Ya se ha dicho que no hay una relación biunívoca entre los tipos de python y los de C. La mayoría de las sorpresas se deberán al uso de un único tipo para los enteros en python o del tipo `intp` en numpy que no es más que un enlace al tipo de entero por defecto de la precisión del procesador, distinto en 32 y en 64 bits.

Este pequeño ejemplo ilustra el uso de las cadenas de texto de un modo un poco surrealista:

```
>>> s=c_char_p('Hello, world!')
>>> s
c_char_p('Hello, world!')
>>> libc.printf('%s\n',s)
Hello, world!
```

¡Es un Hola, mundo en C desde python! Uno puede apreciar la magnitud de la evolución de los lenguajes de programación en las últimas tres décadas con este ejemplo. Realizar esta unión en cualquier otro lenguaje interpretado, incluso en python en sus versiones anteriores no hubiera resultado muy difícil pero no trivial como es el caso.

Arrays

La declaración de un array en ctypes es sencilla gracias a este truco sintáctico:

```
>>> ctypes_array = c_int * 3
>>> ctypes_array
<class 'ctypes._endian.c_long_Array_3'>
```

Sin embargo este comando es sólo la declaración. Para declarar e iniciar el array puede hacerse lo siguiente:

```
>>> ctypes_array = (c_int * 3)(1,2,3)
>>> ctypes_array
<ctypes._endian.c_long_Array_3 object at 0xb7b6d26c>
>>> for element in ctypes_array:
...     print element
...
1
2
3
```

Una pequeña novedad respecto al uso de C es que las fronteras del array existen realmente:

```
>>> ctypes_array[2]
3
>>> ctypes_array[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: invalid index
```

Arrays en ctypes y en numpy

Pero lo realmente interesante es la unión entre los arrays declarados mediante ctypes y numpy. Afortunadamente esta unión es muy sencilla.

Crear un array en numpy desde un array de ctypes

Es tan fácil como:

```
>>> from numpy import *
>>> numpy_array=array(ctypes_array)
>>> numpy_array
array([1, 2, 3])
```

¡Basta con iniciarlo!

Note

Para quienes no tengan demasiada experiencia con el modo en que C trata la memoria: un array no es más que un puntero al que va asociado una cantidad de memoria especificada. En Fortran, en Matlab, en python... Un array es algo más, tiene dimensiones, se queja cuando se llama un elemento que no existe... La consecuencia directa es que en C el uso de las dimensiones no es real, para C sólo existen los vectores puesto que no tiene información alguna del tamaño o las dimensiones.

Como en C, no es lo mismo un escalar que un array. El concepto de almacenamiento es distinto y python es consecuente con ello. Para entenderlo nada mejor que un ejemplo. Supongamos que se desea operar con un array de dimensión cero. Como este array es equivalente a un escalar parece lógico utilizar directamente un escalar:

```
>>> d=c_double(4.23)
>>> numpy_d=array(d)
>>> numpy_d.dtype
dtype('object')
>>> numpy_d/1.35
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for /: 'c_double' and 'float'
```

Como se aprecia, numpy no hace una conversión de tipo de los escalares, mantiene su integraidad como elementos de ctypes. Entonces hay que diferenciar explítitamente entre escalares y arrays de dimensión cero:

```
>>> d=(c_double *1)(4.23)
>>> numpy_d=array(d)
>>> numpy_d.dtype
dtype('float64')
>>> numpy_d/1.35
array([ 3.13333333])
```

Important

Los arrays creados a partir de arrays de ctypes son copias, no comparten almacenamiento. Teniendo en cuenta lo anterior, si se cambia la variable `numpy_d` , la variable d no cambia de valor:

```
>>> numpy_d[0]=1
>>> numpy_d
array([ 1.])
>>> d[0]
4.2300000000000004
```

El comportamiento no cambia al utilizar el flag copy en la llamada a array. Esta consideración es importante en el caso de manipular variables demasiado grandes como para permitir una copia en memoria

Llamar una función desde ctypes

Las funciones como elemento de programa reciben unos argumentos con un tipo determinado. Es lógico que pasar un argumento con un tipo equivocado suele provocar errores o violaciones de segmento. Sin embargo no existe ningún indicio que el error ha sido precisamente en el momento de la llamada, el más trivial, y suele comprobarse tras un largo proceso de debugging.

En ctypes cada función cuenta con el atributo argtypes para especificar los tipos de argumentos que debe recibir cada una. En el caso que, en el momento de utilizarla, reciba algún tipo erróneo python devolverá un error de tipo.

Este es un ejemplo del manual que ilustra perfectamente el funcionamiento del atributo `argtypes`.

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc.printf.argtypes=[c_char_p, c_char_p, c_double]
>>> libc.printf('%s,y %f\n','una cadena',2.3345)
una cadena,y 2.334500
22
>>> libc.printf('%s,y %f\n','una cadena','otra cadena')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ctypes.ArgumentError: argument 3: exceptions.TypeError:
     float expected instead of str instance
```

En el caso que alguno de los argumentos de la función deba pasarse por referencia puede especificarse en los atributos para que se lleve a cabo la correspondiente conversión de tipo. Por ejemplo, si el tercer argumento es en realidad un puntero a un real de doble precisión:

```
>>> libc.printf.argtypes = [c_char_p,c_char_p,POINTER(c_double)]
```

Otra opción sería especificar la llamada por referencia en la propia llamada a la función:

```
>>> libc.printf('%s,y %f\n','una cadena',byref(2.3345))
```

ctypes para numpy

Numpy ha adoptado ctypes para comunicarse con funciones compiladas de forma manual y cuenta con un módulo de adaptación en `numpy.ctypeslib`. Este módulo no sólo se sitúa entre numpy y ctypes sino que además permite pasar arrays como argumentos de funciones enlazadas mediante ctypes. Intentar hacerlo de otra manera puede resultar laborioso y poco efectivo puesto que ctypes asume que los elementos son contiguos en memoria y no tinene ningún control sobre la ordenación de los elementos.

Los dos elementos clave del módulo son el `ctypeslib.ndpointer` y `ctypeslib.load_library` .

El primero sirve para explicitar que uno de los argumentos es un array y el segundo es equivalente a `ctypes.cdll.LoadLibrary` con la diferencia que recibe el argumento necesario de dónde buscar la biblioteca.

Un pequeño wrapper para lapack

Para ilustrar todo lo anterior se procederá a crear un interfaz o wrapper sobre la rutina de lapack `dgesv`. Este ejemplo tiene una utilidad meramente ilustrativa puesto que los wrappers de los que dispone scipy son mucho más completos y funcionales:

```
from ctypes import c_int, POINTER
import numpy as np
from numpy.ctypeslib import load_library,ndpointer
```

```
def dgesv(N,A,B):
    A = np.asfortranarray(A.astype(np.float64))  #1
    B = np.asfortranarray(B.astype(np.float64))

    cN=c_int(N)  #2
    NRHS=c_int(1)
    LDA=c_int(N)
    IPIV=(c_int * N)() #3
    LDB=c_int(N)
    INFO=c_int(1)

    lapack=load_library('liblapack.so','/usr/lib/') #4

    lapack.dgesv_.argtypes=[POINTER(c_int),POINTER(c_int),
                            ndpointer(dtype=np.float64,
                            ndim=2,
                            flags='FORTRAN'),
                            POINTER(c_int), POINTER(c_int),
                            ndpointer(dtype=np.float64,
                                      ndim=2,
                                      flags='FORTRAN'),
                            POINTER(c_int),POINTER(c_int)] #5


    lapack.dgesv_(cN,NRHS,A,LDA,IPIV,B,LDB,INFO) #6
    return B

print dgesv(2,np.array([[1,2],[1,4]]),np.array([[1,3]]))
```

Este pequeño script define la función `dgesv`, un wrapper a la función de lapack que resuelve sistemas de ecuaciones lineales generales en doble precisión. Luego llama la función con los argumentos necesarios

1. Esta línea es la conversión de tipo necesaria para ajustarse a la especificación de argumentos. En vez de no convertir nada y dejar que la llamada a `dgesv_` de un error de tipo. Como se puede apreciar, el coste en código de convertir los argumentos es poco.

2. Los argumentos de entrada son obviamente argumentos de ctypes y deben ser declarados.

3. Este es el modo de declarar un array sin iniciarlo con ningún valor.

4. Se carga la biblioteca a la variable `lapack`

5. Especificación de los argumentos. Las conversiones de tipo anteriores son precisamente para ajustarse a esta especificación. Como todos los argumentos son llamados por referencia es necesario convertirlos a punteros mediante la función `POINTER`.

6. Llamada a la función. Salta a la vista el hecho que en nombre de la función incluya el símbolo `_`. En la jerga de programación se llama *trailing underscore*. Cuando una función incluye este simbolo significa que no está escrita en C sino en fortran con todo lo que ello implica. Las funciones escritas en Fortran pueden llamarse sin problemas en C pero debe tenerse en cuenta que todos los argumentos deben pasarse por referencia.

Al ejecutar el script:

```
>>> python lapackw.py
[[-1.  1.]]
```

TODO. Definir cómo hacer callbacks para poder llamar a la librería sundials.
Many thanks to Stefan van der Walt