

Creado por:

Isabel Maniega

```
In [2]: from IPython.display import Image
```

Paquete Ctypes

Proporciona tipos de datos compatibles con C y permite llamar funciones en DLL o bibliotecas compartidas. Se puede utilizar para encapsular estas bibliotecas en Python puro.

Realizaremos un ejemplo para conocer como funciona realizamos un script en el cual tendremos las siguientes funciones:

clib1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int simple_function(void) {
    static int counter = 0;
    counter++;
    return counter;
}

void add_one_to_string(char *input) {
    int ii = 0;
    for (; ii < strlen(input); ii++) {
        input[ii]++;
    }
}

char * alloc_C_string(void) {
    char* phrase = strdup("I was written in C");
    printf("      C just allocated %p(%ld):  %s\n",
phrase, (long int)phrase, phrase);
    return phrase;
}

void free_C_string(char* ptr) {
    printf("      About to free %p(%ld):  %s\n", ptr,
(long int)ptr, ptr);
    free(ptr);
}
```

- Función `simple_function` simplemente devuelve números de conteo. Cada vez que se la llama, se incrementa el contador y se devuelve ese valor.

- Función `add_one_to_string` agrega uno a cada carácter en una matriz de caracteres que se pasa. La usaremos para hablar sobre las cadenas inmutables de Python y cómo solucionarlas cuando sea necesario.
- Las funciones `alloc_C_string` y `free_C_string` asignan y liberan una cadena en el contexto de C. Esto proporcionará el marco para hablar sobre la gestión de memoria en ctypes.

Por último, necesitamos una forma de convertir este archivo fuente en una biblioteca. Si bien hay muchas herramientas pero prefiero utilizar make, lo utilizo para proyectos como este debido a su bajo consumo de recursos y su ubicuidad. Make está disponible en todos los sistemas similares a Linux.

A continuación, se incluye un fragmento del Makefile que convierte la biblioteca C en un archivo .so:

Makefile

```
clib1.so: clib1.o
    gcc -shared -o libclib1.so clib1.o

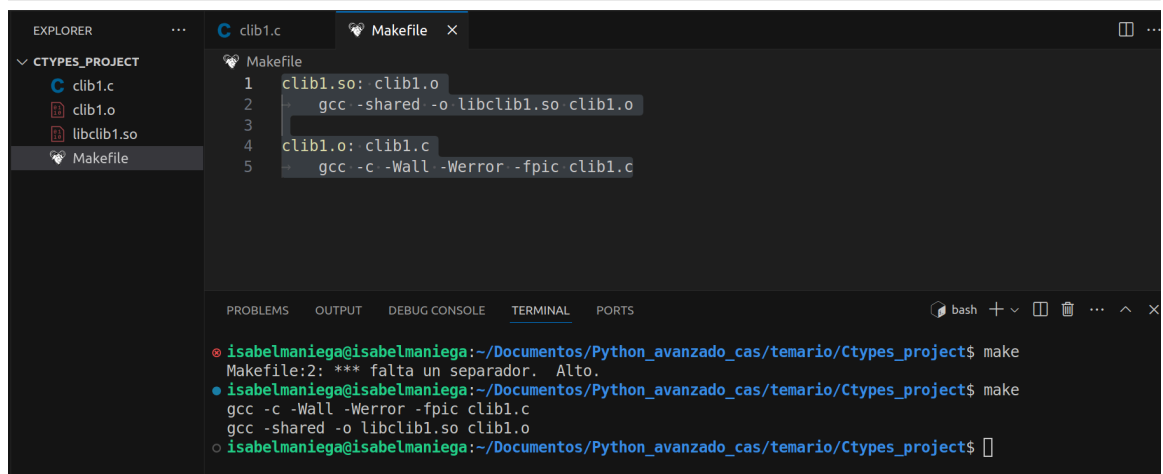
clib1.o: clib1.c
    gcc -c -Wall -Werror -fpic clib1.c
```

El Makefile en el repositorio está configurado para compilar y ejecutar completamente la demostración desde cero; solo necesita ejecutar el siguiente comando en su shell:

make

In [3]: `Image(filename='./images/ctypes_1.png')`

Out[3]:



The screenshot shows a VS Code editor with a file explorer on the left displaying the project structure: `CTYPES_PROJECT` containing `clib1.c`, `clib1.o`, `libclib1.so`, and `Makefile`. The main editor window shows the `Makefile` content, which matches the code provided in the previous block. Below the editor, a terminal window shows the execution of the `make` command. The first attempt fails with the error `Makefile:2: *** falta un separador. Alto.`. The second attempt succeeds, showing the compilation of `clib1.c` into `clib1.o` and the linking of `libclib1.so`.

```
1 clib1.so: clib1.o
2 gcc -shared -o libclib1.so clib1.o
3
4 clib1.o: clib1.c
5 gcc -c -Wall -Werror -fpic clib1.c
```

```
isabelmaniega@isabelmaniega:~/Documentos/Python_avanzado_cas/temario/Ctypes_project$ make
Makefile:2: *** falta un separador. Alto.
isabelmaniega@isabelmaniega:~/Documentos/Python_avanzado_cas/temario/Ctypes_project$ make
gcc -c -Wall -Werror -fpic clib1.c
gcc -shared -o libclib1.so clib1.o
isabelmaniega@isabelmaniega:~/Documentos/Python_avanzado_cas/temario/Ctypes_project$
```

Cargar una biblioteca C con el módulo “ctypes” de Python

Ctypes le permite cargar una biblioteca compartida (“DLL” en Windows) y acceder a métodos directamente desde ella, siempre que tenga cuidado de “ordenar” los datos correctamente.

La forma más básica de esto es:

```
In [4]: import ctypes

# Load the shared library into c types.
libc = ctypes.CDLL("./Ctypes_project/libclib1.so")
```

Llamar a funciones simples con ctypes

Lo bueno de los ctypes es que simplifican mucho las cosas simples. Llamar a una función sin parámetros es trivial. Una vez que haya cargado la biblioteca, la función es solo un método del objeto de biblioteca.

```
In [5]: import ctypes

# Load the shared library into c types.
libc = ctypes.CDLL("./Ctypes_project/libclib1.so")

# Call the C function from the library
counter = libc.simple_function()
counter
```

Out[5]: 1

Recordarás que la función C que estamos llamando devuelve números de conteo como objetos int. Nuevamente, ctypes facilita las cosas fáciles: pasar int funciona sin problemas y hace prácticamente lo que esperas.

Cómo manejar cadenas mutables e inmutables como parámetros de ctypes

Si bien los tipos básicos, int y float, generalmente se ordenan mediante ctypes de manera trivial, las cadenas plantean un problema. En Python, las cadenas son inmutables, lo que significa que no pueden cambiar. Esto produce un comportamiento extraño al pasar cadenas en ctypes.

Para este ejemplo, usaremos la función `add_one_to_string` que se muestra en la biblioteca C anterior. Si la llamamos al pasar una cadena de Python, se ejecuta, pero no modifica la cadena como podríamos esperar. Este código de Python:

```
In [6]: print("Calling C function which tries to modify Python string")
original_string = "starting string"
print("Before:", original_string)

# This call does not change value, even though it tries!
libc.add_one_to_string(original_string)

print("After: ", original_string)
```

Calling C function which tries to modify Python string

Before: starting string

After: starting string

Después de algunas pruebas, se observa que la cadena original no está disponible en absoluto en la función C al hacer esto. La cadena original no cambió, principalmente porque la función C modificó otra memoria, no la cadena. Por lo tanto, la función C no solo no hace lo que quieres, sino que también modifica la memoria que no debería, lo que lleva a posibles problemas de corrupción de memoria.

Si queremos que la función C tenga acceso a la cadena, necesitamos hacer un pequeño trabajo de ordenación por adelantado. Afortunadamente, ctypes también hace que esto sea bastante fácil.

Necesitamos convertir la cadena original a bytes usando str.encode y luego pasar esto al constructor para un ctypes.string_buffer. Los string_buffers son mutables y se pasan a C como un char * como esperaría.

```
In [7]: # The ctypes string buffer IS mutable, however.
print("Calling C function with mutable buffer this time")

# Need to encode the original to get bytes for string_buffer
mutable_string = ctypes.create_string_buffer(str.encode(original_string))

print("Before:", mutable_string.value)
libc.add_one_to_string(mutable_string) # Works!
print("After: ", mutable_string.value)
```

Calling C function with mutable buffer this time

Before: b'starting string'

After: b'tubsujoh!tusjoh'

Tenga en cuenta que string_buffer se imprime como una matriz de bytes en el lado de Python.

Especificación de firmas de funciones en ctypes

Antes de llegar al ejemplo final de este tutorial, debemos hacer una breve pausa y hablar sobre cómo ctypes pasa parámetros y devuelve valores. Como vimos anteriormente, podemos especificar el tipo de retorno si es necesario.

Podemos hacer una especificación similar de los parámetros de la función. Ctypes determinará el tipo del puntero y creará una asignación predeterminada a un tipo de Python, pero eso no siempre es lo que desea hacer. Además, proporcionar una firma de función permite a Python verificar que está pasando los parámetros correctos cuando llama a una función de C; de lo contrario, pueden suceder cosas locas.

Como cada una de las funciones en la biblioteca cargada es en realidad un objeto de Python que tiene sus propias propiedades, especificar el valor de retorno es bastante simple. Para especificar el tipo de retorno de una función, obtiene el objeto de función y establece la propiedad restype de esta manera:

```
In [8]: alloc_func = libc.alloc_C_string
alloc_func.restype = ctypes.POINTER(ctypes.c_char)
```

Conceptos básicos de administración de memoria en ctypes

Una de las grandes características de pasar de C a Python es que ya no es necesario dedicar tiempo a la administración manual de la memoria. La regla de oro al hacer ctypes, o cualquier ordenación entre lenguajes, es que el lenguaje que asigna la memoria también debe liberarla.

En el ejemplo anterior, esto funcionó bastante bien, ya que Python asignó los buffers de cadena que estábamos pasando para poder liberar esa memoria cuando ya no fuera necesaria.

Sin embargo, con frecuencia surge la necesidad de asignar memoria en C y luego pasarla a Python para alguna manipulación. Esto funciona, pero es necesario realizar algunos pasos más para asegurarse de poder pasar el puntero de memoria de vuelta a C para que pueda liberarlo cuando hayamos terminado.

Para este ejemplo, utilizaré estas dos funciones de C, `alloc_C_string` y `free_C_string`. En el código de ejemplo, ambas funciones imprimen el puntero de memoria que están manipulando para dejar en claro lo que está sucediendo.

Como se mencionó anteriormente, necesitamos poder mantener el puntero real a la memoria que `alloc_C_string` asignó para que podamos pasarlo de vuelta a `free_C_string`. Para hacer esto, necesitamos decirle a ctype que `alloc_C_string` debe devolver un `ctypes.POINTER` a un `ctypes.c_char`. Ya vimos eso antes.

Los objetos `ctypes.POINTER` no son demasiado útiles, pero se pueden convertir en objetos que sí lo son. Una vez que convertimos nuestra cadena a un `ctypes.c_char`, podemos acceder a su atributo de valor para obtener los bytes en Python.

Poniendo todo eso junto se ve así:

```
In [9]: alloc_func = libc.alloc_C_string

# This is a ctypes.POINTER object which holds the address of the data
alloc_func.restype = ctypes.POINTER(ctypes.c_char)

print("Allocating and freeing memory in C")
c_string_address = alloc_func()

# Wow we have the POINTER object.
# We should convert that to something we can use
# on the Python side
phrase = ctypes.c_char_p.from_buffer(c_string_address)

print("Bytes in Python {0}".format(phrase.value))
```

```
Allocating and freeing memory in C
Bytes in Python b'I was written in C'
```

Una vez que hemos utilizado los datos que asignamos en C, debemos liberarlos. El proceso es bastante similar, especificando el atributo `argtypes` en lugar de `restype`:

```
In [10]: free_func = libc.free_C_string
free_func.argtypes = [ctypes.POINTER(ctypes.c_char), ]
free_func(c_string_address)
```

```
Out[10]: 0
```

Tipos de datos fundamentales

`ctypes` define una serie de tipos de datos primitivos compatibles con C:

ctypes type	C type	Python type
<code>c_bool</code>	<code>_Bool</code>	<code>bool</code> (1)
<code>c_char</code>	<code>char</code>	Un objeto <code>bytes</code> de 1-caracter
<code>c_wchar</code>	<code>wchar_t</code>	1-caracter unicode string
<code>c_byte</code>	<code>char</code>	<code>int/long</code>
<code>c_ubyte</code>	unsigned char	<code>int/long</code>
<code>c_short</code>	<code>short</code>	<code>int/long</code>
<code>c_ushort</code>	unsigned short	<code>int/long</code>
<code>c_int</code>	<code>int</code>	<code>int/long</code>
<code>c_uint</code>	unsigned int	<code>int/long</code>
<code>c_long</code>	<code>long</code>	<code>int/long</code>
<code>c_ulong</code>	unsigned long	<code>int/long</code>
<code>c_longlong</code>	<code>__int64</code> or long long	<code>int/long</code>
<code>c_ulonglong</code>	unsigned <code>__int64</code> or unsigned long long	<code>int/long</code>
<code>c_size_t</code>	<code>size_t</code>	<code>int</code>
<code>c_ssize_t</code>	<code>ssize_t</code> or <code>Py_ssize_t</code>	<code>int</code>
<code>c_float</code>	<code>float</code>	<code>float</code>
<code>c_double</code>	<code>double</code>	<code>float</code>
<code>c_longdouble</code>	long double	<code>float</code>
<code>c_char_p</code>	<code>char *</code> (NUL terminated)	objeto de <code>bytes</code> o <code>None</code>
<code>c_wchar_p</code>	<code>wchar_t *</code> (NUL terminated)	string or <code>None</code>
<code>c_void_p</code>	<code>void *</code>	<code>int/long</code> or <code>None</code>

Todos estos tipos se pueden crear llamándolos con un inicializador opcional del tipo y valor correctos:

```
In [11]: i = ctypes.c_int(42)
i
```

```
Out[11]: c_int(42)
```

```
In [12]: i.value
```

```
Out[12]: 42
```

Todos estos tipos pueden ser creados llamándolos con un inicializador opcional del tipo y valor correctos:

```
In [13]: s = "Hello, World"
c_s = ctypes.c_wchar_p(s)
c_s
```

```
Out[13]: c_wchar_p(125675094174256)
```

```
In [14]: c_s.value
```

```
Out[14]: 'Hello, World'
```

Asignando un nuevo valor a las instancias de los tipos de punteros `c_char_p`, `c_wchar_p`, y `c_void_p` cambia el lugar de memoria al que apuntan, no el contenido del bloque de memoria (por supuesto que no, porque los objetos de bytes de Python son inmutables):

```
In [15]: c_s.value = "Hi, there"
print(c_s) # the memory location has changed
```

```
c_wchar_p(125675062472128)
```

```
In [16]: print(c_s.value)
print(s)
```

```
Hi, there
Hello, World
```

Estructuras y uniones

Las estructuras y uniones deben derivar de las clases base `Structure` y `Union` que se definen en el módulo `ctypes`. Cada subclase debe definir un atributo `_fields_`. `_fields_` debe ser una lista de 2-tuplas, que contenga un nombre de campo y un tipo de campo.

El tipo de campo debe ser un tipo `ctypes` como `c_int`, o cualquier otro tipo `ctypes` derivado: estructura, unión, matriz, puntero.

Aquí hay un ejemplo simple de una estructura `POINT`, que contiene dos enteros llamados `x` y `y`, y también muestra cómo inicializar una estructura en el constructor:

```
In [17]: from ctypes import *

class POINT(Structure):
    _fields_ = [("x", c_int),
                ("y", c_int)]

point = POINT(10, 20)
point.x, point.y
```

Out[17]: (10, 20)

```
In [18]: point = POINT(y=5)
         point.x, point.y
```

Out[18]: (0, 5)

```
In [19]: POINT(1, 2, 3)
```

```
-----
-
TypeError                                Traceback (most recent call las
t)
Cell In[19], line 1
----> 1 POINT(1, 2, 3)

TypeError: too many initializers
```

Sin embargo, se pueden construir estructuras mucho más complicadas. Una estructura puede contener por sí misma otras estructuras usando una estructura como tipo de campo.

Aquí hay una estructura RECT que contiene dos POINTs llamados upperleft (superior izquierda) y lowerright (abajo a la derecha):

```
In [20]: class RECT(Structure):
         _fields_ = [("upperleft", POINT),
                    ("lowerright", POINT)]

         rc = RECT(point)
         print(rc.upperleft.x, rc.upperleft.y)
         print(rc.lowerright.x, rc.lowerright.y)
```

0 5

0 0

Las estructuras anidadas también pueden ser inicializadas en el constructor de varias maneras:

```
In [21]: r = RECT(POINT(1, 2), POINT(3, 4))
         r = RECT((1, 2), (3, 4))
         # El campo descriptor puede ser recuperado de la class, son útiles para l
         print(POINT.x)
         print(POINT.y)
```

<Field type=c_int, ofs=0, size=4>

<Field type=c_int, ofs=4, size=4>

Arrays

Los arrays son secuencias, que contienen un número fijo de instancias del mismo tipo.

La forma recomendada de crear tipos de arreglos es multiplicando un tipo de dato por un entero positivo:

```
TenPointsArrayType = POINT * 10
```


Aquí hay un ejemplo de un tipo de datos algo artificial, una estructura que contiene 4 POINTs entre otras cosas:

```
In [22]: from ctypes import *

class POINT(Structure):
    _fields_ = (("x", c_int),
                ("y", c_int))

class MyStruct(Structure):
    _fields_ = [("a", c_int),
                ("b", c_float),
                ("point_array", POINT * 4)]

print(len(MyStruct().point_array))
```

4

```
In [23]: TenPointsArrayType = POINT * 10
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0

El código anterior imprime una serie de líneas 0 0, porque el contenido del arreglos se inicializa con ceros.

También se pueden especificar inicializadores del tipo correcto:

```
In [24]: TenIntegers = c_int * 10

ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
print(ii)

<__main__.c_int_Array_10 object at 0x724d00d77740>
```

```
In [25]: for i in ii:
          print(i, end=" ")
```

1 2 3 4 5 6 7 8 9 10

Punteros

Las instancias de puntero se crean llamando a la función pointer() en un tipo ctypes:

```
In [26]: i = c_int(42)
          pi = pointer(i)
```

Las instancias del puntero tienen un atributo `contents` que retorna el objeto al que apunta el puntero, el objeto `i` arriba:

```
In [27]: pi.contents
```

```
Out[27]: c_int(42)
```

Ten en cuenta que ctypes no tiene OOR (original object return), construye un nuevo objeto equivalente cada vez que recuperas un atributo:

```
In [28]: pi.contents is i
```

```
Out[28]: False
```

```
In [29]: pi.contents is pi.contents
```

```
Out[29]: False
```

Asignar otra instancia `c_int` al atributo de contenido del puntero causaría que el puntero apunte al lugar de memoria donde se almacena:

```
In [30]: i = c_int(99)
         pi.contents = i
         pi.contents
```

```
Out[30]: c_int(99)
```

Las instancias de puntero también pueden ser indexadas con números enteros:

```
In [31]: pi[0]
```

```
Out[31]: 99
```

Asignando a un índice entero cambia el valor señalado:

```
In [32]: print(i)
         pi[0] = 22
         print(i)
```

```
c_int(99)
```

```
c_int(22)
```

Creado por:

Isabel Maniega