ctypes tutorial

Contents

- Loading dynamic link libraries
- Accessing functions from loaded dlls
- Calling functions
- Fundamental data types
- Calling functions, continued
- Calling functions with your own custom data types
- Specifying the required argument types (function prototypes)
- Return types
- Passing pointers (or: passing parameters by reference)
- Structures and unions
- Structure/union alignment and byte order
- Bit fields in structures and unions
- Arrays
- Pointers
- Type conversions
- Incomplete Types
- Callback functions
- Accessing values exported from dlls
- Surprises
- Variable-sized data types
- Bugs, ToDo and non-implemented things

Note: The code samples in this tutorial uses `doctest` to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or Mac OS X, they contain doctest directives in comments.

Note: Quite some code samples references the ctypes `c_int` type. This type is an alias to the `c_long` type on 32-bit systems. So, you should not be confused if `c_long` is printed if you would expect `c_int` - they are actually the same type.

Loading dynamic link libraries

`ctypes` exports the `cdll`, and on Windows also `windll` and `oledll` objects to load dynamic link libraries.

You load libraries by accessing them as attributes of these objects. `cdll` loads libraries which export functions using the standard `cdecl` calling convention, while `windll` libraries call functions using the `stdcall` calling convention. `oledll` also uses the `stdcall` calling convention, and assumes the functions return a Windows `HRESULT` error code. The error code is used to automatically raise `WindowsError` Python exceptions when the function call fails.

Here are some examples for Windows, note that `msvcrt` is the MS standard C library containing most standard C functions, and uses the cdecl calling convention:

```
>>> from ctypes import *
>>> print windll.kernel32 # doctest: +WINDOWS
<WinDLL 'kernel32', handle ... at ...>
>>> print cdll.msvcrt # doctest: +WINDOWS
<CDLL 'msvcrt', handle ... at ...>
```

```
>>> libc = cdll.msvcrt # doctest: +WINDOWS
>>>
```

Windows appends the usual '.dll' file suffix automatically.

On Linux, it is required to specify the filename *including* the extension to load a library, so attribute access does not work. Either the `LoadLibrary` method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

```
>>> cdll.LoadLibrary("libc.so.6") # doctest: +LINUX
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")     # doctest: +LINUX
>>> libc                         # doctest: +LINUX
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Accessing functions from loaded dlls

Functions are accessed as attributes of dll objects:

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print windll.kernel32.GetModuleHandleA # doctest: +WINDOWS
<_FuncPtr object at 0x...>
>>> print windll.kernel32.MyOwnFunction # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with an `W` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32 `GetModuleHandle` function, which returns a *module handle* for a given module name, has the following C prototype, and a macro is used to expose one of them as `GetModuleHandle` depending on whether UNICODE is defined or not:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

`windll` does not try to select one of them by magic, you must access the version you need by specifying `GetModuleHandleA` or `GetModuleHandleW` explicitely, and then call it with normal strings or unicode strings respectively.

Sometimes, dlls export functions with names which aren't valid Python identifiers, like `"??2@YAPAXI@Z"`. In this case you have to use `getattr` to retrieve the function:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z") # doctest: +WINDOWS
<_FuncPtr object at 0x...>
>>>
```

On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```
>>> cdll.kernel32[1] # doctest: +WINDOWS
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0] # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

Calling functions

You can call these functions like any other Python callable. This example uses the `time()` function, which returns system time in seconds since the UNIX epoch, and the `GetModuleHandleA()` function, which returns a win32 module handle.

This example calls both functions with a NULL pointer (`None` should be used as the NULL pointer):

```
>>> print libc.time(None) # doctest: +SKIP
1150640792
>>> print hex(windll.kernel32.GetModuleHandleA(None)) # doctest: +WINDOWS
0x1d000000
>>>
```

`ctypes` tries to protect you from calling functions with the wrong number of arguments or the wrong calling convention. Unfortunately this only works on Windows. It does this by examining the stack after the function returns, so although an error is raised the function *has* been called:

```
>>> windll.kernel32.GetModuleHandleA() # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>> windll.kernel32.GetModuleHandleA(0, 0) # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

The same exception is raised when you call an `stdcall` function with the `cdecl` calling convention, or vice versa:

```
>>> cdll.kernel32.GetModuleHandleA(None) # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf("spam") # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, `ctypes` uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```
>>> windll.kernel32.GetModuleHandleA(32) # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
WindowsError: exception: access violation reading 0x00000020
>>>
```

There are, however, enough ways to crash Python with `ctypes`, so you should be careful anyway.

`None`, integers, longs, byte strings and unicode strings are the only native Python objects that can directly be used as parameters in these function calls. `None` is passed as a C `NULL` pointer, byte strings and unicode strings are passed as pointer to the memory block that contains their data (`char *` or `wchar_t *`). Python integers and Python longs are passed as the platforms default C `int` type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about `ctypes` data types.

Fundamental data types

`ctypes` defines a number of primitive C compatible data types :

ctypes type	C type	Python type
`c_char`	`char`	1-character string
`c_wchar`	`wchar_t`	1-character unicode string
`c_byte`	`char`	int/long
`c_ubyte`	`unsigned char`	int/long
`c_short`	`short`	int/long
`c_ushort`	`unsigned short`	int/long
`c_int`	`int`	int/long
`c_uint`	`unsigned int`	int/long
`c_long`	`long`	int/long
`c_ulong`	`unsigned long`	int/long
`c_longlong`	`__int64` or `long long`	int/long
`c_ulonglong`	`unsigned __int64` or `unsigned long long`	int/long

ctypes type	C type	Python type
c_float	float	float
c_double	double	float
c_char_p	char * (NUL terminated)	string or None
c_wchar_p	wchar_t * (NUL terminated)	unicode or None
c_void_p	void *	int/long or None

All these types can be created by calling them with an optional initializer of the correct type and value:

```
>>> c_int()
c_long(0)
>>> c_char_p("Hello, World")
c_char_p('Hello, World')
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Since these types are mutable, their value can also be changed afterwards:

```
>>> i = c_int(42)
>>> print i
c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
>>>
```

Assigning a new value to instances of the pointer types `c_char_p`, `c_wchar_p`, and `c_void_p` changes the *memory location* they point to, *not the contents* of the memory block (of course not, because Python strings are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_char_p(s)
>>> print c_s
c_char_p('Hello, World')
>>> c_s.value = "Hi, there"
>>> print c_s
c_char_p('Hi, there')
>>> print s                 # first string is unchanged
Hello, World
>>>
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a `create_string_buffer` function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property, if you want to access it as NUL terminated string, use the `string` property:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)      # create a 3 byte buffer, initialized to NUL bytes
```

```
>>> print sizeof(p), repr(p.raw)
3 '\x00\x00\x00'
>>> p = create_string_buffer("Hello")      # create a buffer containing a NUL terminated string
>>> print sizeof(p), repr(p.raw)
6 'Hello\x00'
>>> print repr(p.value)
'Hello'
>>> p = create_string_buffer("Hello", 10)  # create a 10 byte buffer
>>> print sizeof(p), repr(p.raw)
10 'Hello\x00\x00\x00\x00\x00'
>>> p.value = "Hi"
>>> print sizeof(p), repr(p.raw)
10 'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

The `create_string_buffer` function replaces the `c_buffer` function (which is still available as an alias), as well as the `c_string` function from earlier ctypes releases. To create a mutable memory block containing unicode characters of the C type `wchar_t` use the `create_unicode_buffer` function.

Calling functions, continued

Note that printf prints to the real standard output channel, *not* to `sys.stdout`, so these examples will only work at the console prompt, not from within *IDLE* or *PythonWin*:

```
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
>>> printf("Hello, %S", u"World!")
Hello, World!
13
>>> printf("%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf("%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

As has been mentioned before, all Python types except integers, strings, and unicode strings have to be wrapped in their corresponding `ctypes` type, so that they can be converted to the required C data type:

```
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
Integer 1234, double 3.1400001049
31
>>>
```

Calling functions with your own custom data types

You can also customize `ctypes` argument conversion to allow instances of your own classes be used as function arguments. `ctypes` looks for an `_as_parameter_` attribute and uses this as the function argument. Of course, it

must be one of integer, string, or unicode:

```
>>> class Bottles(object):
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf("%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

If you don't want to store the instance's data in the `_as_parameter_` instance variable, you could define a `property` which makes the data avaiblable.

Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the `argtypes` attribute.

`argtypes` must be a sequence of C data types (the `printf` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf("String '%s', Int %d, Double %f\n", "Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```
>>> printf("%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf("%s %d %f", "X", 2, 3)
X 2 3.00000012
12
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a `from_param` class method for them to be able to use them in the `argtypes` sequence. The `from_param` class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, it's `_as_parameter_` attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, unicode, a `ctypes` instance, or something having the `_as_parameter_` attribute.

Return types

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the `restype` attribute of the function object.

Here is a more advanced example, it uses the `strchr` function, which expects a string pointer and a char, and returns a pointer to a string:

```
>>> strchr = libc.strchr
>>> strchr("abcdef", ord("d")) # doctest: +SKIP
8059983
>>> strchr.restype = c_char_p # c_char_p is a pointer to a string
>>> strchr("abcdef", ord("d"))
'def'
>>> print strchr("abcdef", ord("x"))
None
>>>
```

If you want to avoid the `ord("x")` calls above, you can set the `argtypes` attribute, and the second argument will be converted from a single character Python string into a C char:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr("abcdef", "d")
'def'
>>> strchr("abcdef", "def")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print strchr("abcdef", "x")
None
>>> strchr("abcdef", "d")
'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the `restype` attribute, if the foreign function returns an integer. The callable will be called with the `integer` the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA # doctest: +WINDOWS
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle # doctest: +WINDOWS
>>> GetModuleHandle(None) # doctest: +WINDOWS
486539264
>>> GetModuleHandle("something silly") # doctest: +WINDOWS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in ValidHandle
```

```
WindowsError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` is a function which will call Windows `FormatMessage()` api to get the string representation of an error code, and *returns* an exception. `WinError` takes an optional error code parameter, if no one is used, it calls `GetLastError()` to retrieve it.

Please note that a much more powerful error checking mechanism is available through the `errcheck` attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a *pointer* to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as *passing parameters by reference*.

`ctypes` exports the `byref` function which is used to pass parameters by reference. The same effect can be achieved with the `pointer` function, although `pointer` does a lot more work since it constructs a real pointer object, so it is faster to use `byref` if you don't need the pointer object in Python itself:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 ''
>>> libc.sscanf("1 3.14 Hello", "%d %f %s",
...             byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.1400001049 'Hello'
>>>
```

Structures and unions

Structures and unions must derive from the `Structure` and `Union` base classes which are defined in the `ctypes` module. Each subclass must define a `_fields_` attribute. `_fields_` must be a list of *2-tuples*, containing a *field name* and a *field type*.

The field type must be a `ctypes` type like `c_int`, or any other derived `ctypes` type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two integers named `x` and `y`, and also shows how to initialize a structure in the constructor:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
```

```
>>> print point.x, point.y
10 20
>>> point = POINT(y=5)
>>> print point.x, point.y
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: too many initializers
>>>
```

You can, however, build much more complicated structures. Structures can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named `upperleft` and `lowerright`

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print rc.upperleft.x, rc.upperleft.y
0 5
>>> print rc.lowerright.x, rc.lowerright.y
0 0
>>>
```

Nested structures can also be initialized in the constructor in several ways:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Fields descriptors can be retrieved from the *class*, they are useful for debugging because they can provide useful information:

```
>>> print POINT.x
<Field type=c_long, ofs=0, size=4>
>>> print POINT.y
<Field type=c_long, ofs=4, size=4>
>>>
```

Structure/union alignment and byte order

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behaviour be specifying a `_pack_` class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what `#pragma pack(n)` also does in MSVC.

`ctypes` uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the BigEndianStructure, LittleEndianStructure, BigEndianUnion, and LittleEndianUnion base classes. These classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the `_fields_` tuples:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print Int.first_16
<Field type=c_long, ofs=0:0, bits=16>
>>> print Int.second_16
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```
TenPointsArrayType = POINT * 10
```

Here is an example of an somewhat artifical data type, a structure containing 4 POINTs among other stuff:

```
>>> 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
>>>
```

Instances are created in the usual way, by calling the class:

```
arr = TenPointsArrayType()
for pt in arr:
    print pt.x, pt.y
```

The above code print a series of `0 0` lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print ii
<c_long_Array_10 object at 0x...>
>>> for i in ii: print i,
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Pointers

Pointer instances are created by calling the `pointer` function on a `ctypes` type:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Pointer instances have a `contents` attribute which returns the object to which the pointer points, the `i` object above:

```
>>> pi.contents
c_long(42)
>>>
```

Note that `ctypes` does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Assigning another `c_int` instance to the pointer's contents attribute would cause the pointer to point to the memory location where this is stored:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Pointer instances can also be indexed with integers:

```
>>> pi[0]
99
```

```
>>>
```

Assigning to an integer index changes the pointed to value:

```
>>> print i
c_long(99)
>>> pi[0] = 22
>>> print i
c_long(22)
>>>
```

It is also possible to use indexes different from 0, but you must know what you're doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you *know* that the pointer actually points to an array instead of a single item.

Behind the scenes, the `pointer` function does more than simply create pointer instances, it has to create pointer *types* first. This is done with the `POINTER` function, which accepts any `ctypes` type, and returns a new type:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Calling the pointer type without an argument creates a `NULL` pointer. `NULL` pointers have a `False` boolean value:

```
>>> null_ptr = POINTER(c_int)()
>>> print bool(null_ptr)
False
>>>
```

`ctypes` checks for `NULL` when dereferencing pointers (but dereferencing non-`NULL` pointers would crash Python):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Type conversions

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the `argtypes` list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of c_int:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print bar.values[i]
...
1
2
3
>>>
```

To set a POINTER type field to `NULL`, you can assign `None`:

```
>>> bar.values = None
>>>
```

XXX list other conversions...

Sometimes you have instances of incompatible types. In `C`, you can cast one type into another type. `ctypes` provides a `cast` function which can be used in the same way. The `Bar` structure defined above accepts `POINTER(c_int)` pointers or `c_int` arrays for its `values` field, but not instances of other types:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

For these cases, the `cast` function is handy.

The `cast` function can be used to cast a ctypes instance into a pointer to a different ctypes data type. `cast` takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

So, `cast` can be used to assign to the `values` field of `Bar` the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print bar.values[0]
```

```
0
>>>
```

Incomplete Types

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```
struct cell; /* forward declaration */

struct {
    char *name;
    struct cell *next;
} cell;
```

The straightforward translation into ctypes code would be this, but it does not work:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new `class cell` is not available in the class statement itself. In `ctypes`, we can define the `cell` class and set the `_fields_` attribute later, after the class statement:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Lets try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print p.name,
...     p = p.next[0]
...
```

```
foo bar foo bar foo bar foo bar
>>>
```

Callback functions

`ctypes` allows to create C callable function pointers from Python callables. These are sometimes called *callback functions*.

First, you must create a class for the callback function, the class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The CFUNCTYPE factory function creates types for callback functions using the normal cdecl calling convention, and, on Windows, the WINFUNCTYPE factory function creates types for callback functions using the stdcall calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library's `qsort` function, this is used to sort items with the help of a callback function. `qsort` will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer else.

So our callback function receives pointers to integers, and must return an integer. First we create the `type` for the callback function:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

For the first implementation of the callback function, we simply print the arguments we get, and return 0 (incremental development ;-):

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a, b
...     return 0
...
>>>
```

Create the C callable callback:

```
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

And we're ready to go:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +WINDOWS
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
>>>
```

We know how to access the contents of a pointer, so lets redefine our callback:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

Here is what we get on Windows:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +WINDOWS
py_cmp_func 7 1
py_cmp_func 33 1
py_cmp_func 99 1
py_cmp_func 5 1
py_cmp_func 7 5
py_cmp_func 33 5
py_cmp_func 99 5
py_cmp_func 7 99
py_cmp_func 33 99
py_cmp_func 7 33
>>>
```

It is funny to see that on linux the sort function seems to work much more efficient, it is doing less comparisons:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func) # doctest: +LINUX
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

Ah, we're nearly done! The last step is to actually compare the two items and return a useful result:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return a[0] - b[0]
...
>>>
```

Final run on Windows:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func)) # doctest: +WINDOWS
py_cmp_func 33 7
py_cmp_func 99 33
py_cmp_func 5 99
py_cmp_func 1 99
py_cmp_func 33 7
py_cmp_func 1 33
py_cmp_func 5 33
py_cmp_func 5 7
py_cmp_func 1 7
py_cmp_func 5 1
>>>
```

and on Linux:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func)) # doctest: +LINUX
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

It is quite interesting to see that the Windows `qsort` function needs more comparisons than the linux version!

As we can easily check, our array sorted now:

```
>>> for i in ia: print i,
...
1 5 7 33 99
>>>
```

Important note for callback functions:

Make sure you keep references to CFUNCTYPE objects as long as they are used from C code. `ctypes` doesn't, and if you don't, they may be garbage collected, crashing your program when a callback is made.

Accessing values exported from dlls

Sometimes, a dll not only exports functions, it also exports variables. An example in the Python library itself is the `Py_OptimizeFlag`, an integer set to 0, 1, or 2, depending on the `-O` or `-OO` flag given on startup.

`ctypes` can access values like this with the `in_dll` class methods of the type. `pythonapi` is a predefined symbol giving access to the Python C api:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print opt_flag
c_long(0)
>>>
```

If the interpreter would have been started with `-O`, the sample would have printed `c_long(1)`, or `c_long(2)` if `-OO` would have been specified.

An extended example which also demonstrates the use of pointers accesses the `PyImport_FrozenModules` pointer exported by Python.

Quoting the Python docs: *This pointer is initialized to point to an array of ``struct _frozen`` records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.*

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with `ctypes`:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

We have defined the `struct _frozen` data type, so we can get the pointer to the table:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

Since `table` is a `pointer` to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the NULL entry:

```
>>> for item in table:
...    print item.name, item.size
...    if item.name is None:
...        break
...
__hello__ 104
__phello__ -104
__phello__.spam 104
None 0
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not wellknown, it is only used for testing. Try it out with `import __hello__` for example.

Surprises

There are some edges in `ctypes` where you may be expect something else than what actually happens.

Consider the following example:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
3 4 3 4
>>>
```

Hm. We certainly expected the last statement to print `3 4 1 2`. What happended? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Note that `temp0` and `temp1` are objects still using the internal buffer of the `rc` object above. So executing `rc.a = temp0` copies the buffer contents of `temp0` into `rc` 's buffer. This, in turn, changes the contents of `temp1`. So, the last assignment `rc.b = temp1`, doesn't have the expected effect.

Keep in mind that retrieving subobjects from Structure, Unions, and Arrays doesn't *copy* the subobject, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave different from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
>>>
```

Why is it printing `False`? ctypes instances are objects containing a memory block plus some descriptors accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the `contents` of the object is stored. Accessing the contents again constructs a new Python each time!

Variable-sized data types

`ctypes` provides some support for variable-sized arrays and structures (this was added in version 0.9.9.7).

The `resize` function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a `ValueError` is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print sizeof(short_array)
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

Another way to use variable-sized data types with `ctypes` is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.

Bugs, ToDo and non-implemented things

Enumeration types are not implemented. You can do it easily yourself, using `c_int` as the base class.

`long double` is not implemented.