

15.17. ctypes — A foreign function library for Python

New in version 2.5.

`ctypes` is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

15.17.1. ctypes tutorial

Note: The code samples in this tutorial use `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: Some code samples reference the ctypes `c_int` type. This type is an alias for 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.

15.17.1.1. Loading dynamic link libraries

`ctypes` exports the `cdll`, and on Windows `windll` and `oledll` objects, for loading 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 a `WindowsError` exception 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
<WinDLL 'kernel32', handle ... at ...>
>>> print cdll.msvcrt
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

>>>

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 can not be used to load libraries. 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")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
```

>>>

```
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

15.17.1.2. 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
<_FuncPtr object at 0x...>
>>> print windll.kernel32.MyOwnFunction
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` explicitly, and then call it with 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")
<_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]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
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
>>>
```

15.17.1.3. 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)
1150640792
>>> print hex(windll.kernel32.GetModuleHandleA(None))
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()
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)
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)
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")
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)
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.

15.17.1.4. Fundamental data types

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

ctypes type	C type	Python type
<code>c_bool</code>	<code>_Bool</code>	<code>bool</code> (1)
<code>c_char</code>	<code>char</code>	1-character string
<code>c_wchar</code>	<code>wchar_t</code>	1-character unicode string
<code>c_byte</code>	<code>char</code>	<code>int/long</code>
<code>c_ubyte</code>	<code>unsigned char</code>	<code>int/long</code>
<code>c_short</code>	<code>short</code>	<code>int/long</code>
<code>c_ushort</code>	<code>unsigned short</code>	<code>int/long</code>
<code>c_int</code>	<code>int</code>	<code>int/long</code>
<code>c_uint</code>	<code>unsigned int</code>	<code>int/long</code>
<code>c_long</code>	<code>long</code>	<code>int/long</code>
<code>c_ulong</code>	<code>unsigned long</code>	<code>int/long</code>
<code>c_longlong</code>	<code>__int64</code> Or <code>long long</code>	<code>int/long</code>
<code>c_ulonglong</code>	<code>unsigned __int64</code> Or <code>unsigned long long</code>	<code>int/long</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>	<code>long double</code>	<code>float</code>
<code>c_char_p</code>	<code>char *</code> (NUL terminated)	<code>string</code> or <code>None</code>
<code>c_wchar_p</code>	<code>wchar_t *</code> (NUL terminated)	<code>unicode</code> or <code>None</code>
<code>c_void_p</code>	<code>void *</code>	<code>int/long</code> or <code>None</code>

1. The constructor accepts any object with a truth value.

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
Hello, World
>>>                                     # first string is unchanged
```

>>>

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 `value` 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'
>>>
```

>>>

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.

15.17.1.5. 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\n", u"World!")
Hello, World!
14
>>> 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))
An int 1234, a double 3.140000
31
>>>
```

15.17.1.6. 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 available.

15.17.1.7. 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\n", "X", 2, 3)
X 2 3.000000
13
>>>
```

```
>>>
```

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, its `_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 an object with an `_as_parameter_` attribute.

15.17.1.8. 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"))
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
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
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.

15.17.1.9. 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'
>>>
```

15.17.1.10. 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. A structure 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))
```

Field [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>
>>>
```

Warning: `ctypes` does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it's not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

15.17.1.11. 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 behavior by 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.

15.17.1.12. 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>
>>>
```

15.17.1.13. 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 a somewhat artificial 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
>>>
```

15.17.1.14. 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 invalid 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
>>>
```

15.17.1.15. 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
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as `POINTER(c_int)`) in `argtypes`, an object of the pointed type (`c_int` in this case) can be passed to the function. `ctypes` will apply the required `byref()` conversion in this case automatically.

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

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

```
>>>
```

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
>>>
```

```
>>>
```

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

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
>>>
```

15.17.1.17. Callback functions

`ctypes` allows creating 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)
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)
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 efficiently, it is doing less comparisons:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
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))
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))
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 is sorted now:

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

>>>

Note: 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.

Also, note that if the callback function is called in a thread created outside of Python's control (e.g. by the foreign code that calls the callback), `ctypes` creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with `threading.local` will *not* survive across different callbacks, even when those calls are made from the same C thread.

15.17.1.18. Accessing values exported from dlls

Some shared libraries not only export functions, they also export 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 well known, it is only used for testing. Try it out with `import __hello__` for example.

15.17.1.19. Surprises

There are some edge cases in `ctypes` where you might expect something other 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 happened? 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 sub-objects from Structure, Unions, and Arrays doesn't *copy* the sub-object, 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 object each time!

15.17.1.20. Variable-sized data types

`ctypes` provides some support for variable-sized arrays and structures.

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.

15.17.2. ctypes reference

15.17.2.1. Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler does (on platforms with several versions of a shared library the most recent should be loaded), while the `ctypes` library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.find_library(name)`

Try to find a library and return a pathname. *name* is the library name without any prefix like *lib*, suffix like *.so*, *.dylib* or version number (this is the form used for the posix linker option *-l*). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, `find_library()` tries to run external programs (`/sbin/ldconfig`, `gcc`, and `objdump`) to find the library file. It returns the filename of the library file. Here are some examples:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

```
>>>
```

On OS X, `find_library()` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, `find_library()` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with `ctypes`, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using `find_library()` to locate the library at runtime.

15.17.2.2. Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

```
class ctypes.CDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
use_last_error=False)
```

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

```
class ctypes.OleDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
use_last_error=False)
```

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific `HRESULT` code. `HRESULT` values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an `WindowsError` is automatically raised.

```
class ctypes.WinDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
use_last_error=False)
```

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

On Windows CE only the standard calling convention is used, for convenience the `WinDLL` and `OleDLL` use the standard calling convention on this platform.

The Python [global interpreter lock](#) is released before calling any function exported by these libraries, and reacquired afterwards.

`class ctypes.PyDLL(name, mode=DEFAULT_MODE, handle=None)`

Instances of this class behave like `CDLL` instances, except that the Python GIL is *not* released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platforms `dlopen` or `LoadLibrary` function is used to load the library into the process, and to get a handle to it.

The `mode` parameter can be used to specify how the library is loaded. For details, consult the `dlopen(3)` manpage. On Windows, `mode` is ignored. On posix systems, `RTLD_NOW` is always added, and is not configurable.

The `use_errno` parameter, when set to true, enables a ctypes mechanism that allows accessing the system `errno` error number in a safe way. ctypes maintains a thread-local copy of the systems `errno` variable; if you call foreign functions created with `use_errno=True` then the `errno` value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function `ctypes.get_errno()` returns the value of the ctypes private copy, and the function `ctypes.set_errno()` changes the ctypes private copy to a new value and returns the former value.

The `use_last_error` parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the `GetLastError()` and `SetLastError()` Windows API functions; `ctypes.get_last_error()` and `ctypes.set_last_error()` are used to request and change the ctypes private copy of the windows error code.

New in version 2.6: The `use_last_error` and `use_errno` optional parameters were added.

`ctypes.RTLD_GLOBAL`

Flag to use as `mode` parameter. On platforms where this flag is not available, it is defined as the integer zero.

`ctypes.RTLD_LOCAL`

Flag to use as `mode` parameter. On platforms where this is not available, it is the same as `RTLD_GLOBAL`.

`ctypes.DEFAULT_MODE`

The default mode which is used to load shared libraries. On OSX 10.3, this is `RTLD_GLOBAL`, otherwise it is the same as `RTLD_LOCAL`.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

```
>>>
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

`PyDLL._handle`

The system handle used to access the library.

`PyDLL._name`

The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the `LibraryLoader` class, either by calling the `LoadLibrary()` method, or by retrieving the library as attribute of the loader instance.

`class ctypes.LibraryLoader(dlltype)`

Class which loads shared libraries. *dlltype* should be one of the `CDLL`, `PyDLL`, `WinDLL`, or `OleDLL` types.

`__getattr__()` has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

`LoadLibrary(name)`

Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

`ctypes.cdll`

Creates `CDLL` instances.

`ctypes.windll`

Windows only: Creates `WinDLL` instances.

`ctypes.oledll`

Windows only: Creates `OleDLL` instances.

`ctypes.pydll`

Creates `PyDLL` instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

`ctypes.pythonapi`

An instance of `PyDLL` that exposes Python C API functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.

15.17.2.3. Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

class ctypes._FuncPtr

Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

restype

Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as `restype` and assign a callable to the `errcheck` attribute.

argtypes

Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the `from_param()` class method of the items in the `argtypes` tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a `c_char_p` item in the `argtypes` tuple will convert a unicode string passed as argument into a byte string using ctypes conversion rules.

New: It is now possible to put items in `argtypes` which are not ctypes types, but each item must have a `from_param()` method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck

Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable(*result*, *func*, *arguments*)

result is what the foreign function returns, as specified by the `restype` attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

exception `ctypes.ArgumentError`

This exception is raised when a foreign function call cannot convert one of the passed arguments.

15.17.2.4. Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function.

`ctypes.CFUNCTYPE(restype, *argtypes, use_errno=False, use_last_error=False)`

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If *use_errno* is set to true, the `ctypes` private copy of the system `errno` variable is exchanged with the real `errno` value before and after the call; *use_last_error* does the same for the Windows error code.

Changed in version 2.6: The optional *use_errno* and *use_last_error* parameters were added.

`ctypes.WINFUNCTYPE(restype, *argtypes, use_errno=False, use_last_error=False)`

Windows only: The returned function prototype creates functions that use the `stdcall` calling convention, except on Windows CE where `WINFUNCTYPE()` is the same as `CFUNCTYPE()`. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

`ctypes.PYFUNCTYPE(restype, *argtypes)`

The returned function prototype creates functions that use the Python calling convention. The function will *not* release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

prototype(address)

Returns a foreign function at the specified address which must be an integer.

prototype(callable)

Create a C callable function (a callback function) from a Python *callable*.

prototype(func_spec[, paramflags])

Returns a foreign function exported by a shared library. *func_spec* must be a 2-tuple (*name_or_ordinal*, *library*). The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])

Returns a foreign function that will call a COM method. *vtbl_index* is the index into the virtual function table, a small non-negative integer. *name* is name of the COM method. *iid* is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the *argtypes* tuple.

The optional *paramflags* parameter creates foreign function wrappers with much more functionality than the features described above.

paramflags must be a tuple of the same length as *argtypes*.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

- 1
Specifies an input parameter to the function.
- 2
Output parameter. The foreign function fills in a value.
- 4
Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows `MessageBoxA` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```
WINUSERAPI int WINAPI
MessageBoxA(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType);
```

Here is the wrapping with `ctypes`:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCSTR, LPCSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", None), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxA", windll.user32), paramflags)
>>>
```

The MessageBox foreign function can now be called in these ways:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
>>>
```

A second example demonstrates output parameters. The win32 `GetWindowRect` function retrieves the dimensions of a specified window by copying them into `RECT` structure that the caller has to supply. Here is the C declaration:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
    HWND hWnd,
    LPRECT lpRect);
```

Here is the wrapping with `ctypes`:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the `GetWindowRect` function now returns a `RECT` instance, when called.

Output parameters can be combined with the `errcheck` protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the `errcheck` function returns the argument tuple it receives unchanged, `ctypes` continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

```
>>>
```

15.17.2.5. Utility functions

`ctypes.addressof(obj)`

Returns the address of the memory buffer as integer. *obj* must be an instance of a ctypes type.

`ctypes.alignment(obj_or_type)`

Returns the alignment requirements of a ctypes type. *obj_or_type* must be a ctypes type or instance.

`ctypes.byref(obj[, offset])`

Returns a light-weight pointer to *obj*, which must be an instance of a ctypes type. *offset* defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code:

```
((char *)&obj) + offset
```

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

New in version 2.6: The *offset* optional argument was added.

`ctypes.cast(obj, type)`

This function is similar to the cast operator in C. It returns a new instance of *type* which points to the same memory block as *obj*. *type* must be a pointer type, and *obj* must be an object that can be interpreted as a pointer.

`ctypes.create_string_buffer(init_or_size[, size])`

This function creates a mutable character buffer. The returned object is a ctypes array of `c_char`.

init_or_size must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

If the first parameter is a unicode string, it is converted into an 8-bit string according to ctypes conversion rules.

`ctypes.create_unicode_buffer(init_or_size[, size])`

This function creates a mutable unicode character buffer. The returned object is a ctypes array of `c_wchar`.

init_or_size must be an integer which specifies the size of the array, or a unicode string which will be used to initialize the array items.

If a unicode string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

If the first parameter is an 8-bit string, it is converted into a unicode string according to ctypes conversion rules.

`ctypes.DllCanUnloadNow()`

Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the `DllCanUnloadNow` function that the `_ctypes` extension dll exports.

`ctypes.DllGetClassObject()`

Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the `DllGetClassObject` function that the `_ctypes` extension dll exports.

`ctypes.util.find_library(name)`

Try to find a library and return a pathname. *name* is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

Changed in version 2.6: Windows only: `find_library("m")` Or `find_library("c")` return the result of a call to `find_msvcr()`.

`ctypes.util.find_msvcr()`

Windows only: return the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

New in version 2.6.

`ctypes.FormatError([code])`

Windows only: Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows api function `GetLastError`.

`ctypes.GetLastError()`

Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows *GetLastError()* function directly, it does not return the ctypes-private copy of the error code.

ctypes.**get_errno()**

Returns the current value of the ctypes-private copy of the system `errno` variable in the calling thread.

New in version 2.6.

ctypes.**get_last_error()**

Windows only: returns the current value of the ctypes-private copy of the system `LastError` variable in the calling thread.

New in version 2.6.

ctypes.**memmove(dst, src, count)**

Same as the standard C memmove library function: copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

ctypes.**memset(dst, c, count)**

Same as the standard C memset library function: fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

ctypes.**POINTER(type)**

This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

ctypes.**pointer(obj)**

This function creates a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

ctypes.**resize(obj, size)**

This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

ctypes.**set_conversion_mode(encoding, errors)**

This function sets the rules that ctypes objects use when converting between 8-bit strings and unicode strings. *encoding* must be a string specifying an encoding, like 'utf-8' or 'mbcs', *errors* must be a string specifying the error handling on encoding/decoding errors. Examples of possible values are "strict", "replace", or "ignore".

`set_conversion_mode()` returns a 2-tuple containing the previous conversion rules. On windows, the initial conversion rules are ('mbcs', 'ignore'), on other systems ('ascii', 'strict').

ctypes.**set_errno(value)**

Set the current value of the ctypes-private copy of the system `errno` variable in the calling thread to *value* and return the previous value.

New in version 2.6.

ctypes.**set_last_error**(*value*)

Windows only: set the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to *value* and return the previous value.

New in version 2.6.

ctypes.**sizeof**(*obj_or_type*)

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

ctypes.**string_at**(*address*[, *size*])

This function returns the string starting at memory address *address*. If *size* is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

ctypes.**WinError**(*code=None*, *descr=None*)

Windows only: this function is probably the worst-named thing in ctypes. It creates an instance of `WindowsError`. If *code* is not specified, `GetLastError` is called to determine the error code. If *descr* is not specified, `FormatError()` is called to get a textual description of the error.

ctypes.**wstring_at**(*address*[, *size*])

This function returns the wide character string starting at memory address *address* as unicode string. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

15.17.2.6. Data types

class ctypes.**_CData**

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the `addressof()` helper function. Another instance variable is exposed as `_objects`; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the `metaclass`):

from_buffer(*source*[, *offset*])

This method returns a ctypes instance that shares the buffer of the *source* object. The *source* object must support the writeable buffer interface. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

New in version 2.6.

from_buffer_copy(*source*[, *offset*])

This method creates a ctypes instance, copying the buffer from the *source* object buffer which must be readable. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

New in version 2.6.

from_address(*address*)

This method returns a ctypes type instance using the memory specified by *address* which must be an integer.

from_param(*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's `argtypes` tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns *obj* if that is an instance of the type. Some types accept other objects as well.

in_dll(*library*, *name*)

This method returns a ctypes type instance exported by a shared library. *name* is the name of the symbol that exports the data, *library* is the loaded shared library.

Common instance variables of ctypes data types:

_b_base_

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The `_b_base_` read-only member is the root ctypes object that owns the memory block.

_b_needsfree_

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects

This member is either `None` or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

15.17.2.7. Fundamental data types

class ctypes._SimpleCData

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. `_SimpleCData` is a subclass of `_CData`, so it inherits their methods and attributes.

Changed in version 2.6: ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character string, for character pointer types it is a Python string or unicode string.

When the `value` attribute is retrieved from a ctypes instance, usually a new object is returned each time. `ctypes` does *not* implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a `restype` of `c_char_p`, you will always receive a Python string, *not* a `c_char_p` instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions `restype` is a subclass of `c_void_p`, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the `value` attribute.

These are the fundamental ctypes data types:

class ctypes.c_byte

Represents the C `signed char` datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_char

Represents the C `char` datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_char_p

Represents the C `char *` datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a string.

class ctypes.c_double

Represents the C `double` datatype. The constructor accepts an optional float initializer.

class ctypes.c_longdouble

Represents the C `long double` datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to `c_double`.

New in version 2.6.

class ctypes.c_float

Represents the C `float` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_int**

Represents the C `signed int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to `c_long`.

class ctypes.**c_int8**

Represents the C 8-bit `signed int` datatype. Usually an alias for `c_byte`.

class ctypes.**c_int16**

Represents the C 16-bit `signed int` datatype. Usually an alias for `c_short`.

class ctypes.**c_int32**

Represents the C 32-bit `signed int` datatype. Usually an alias for `c_int`.

class ctypes.**c_int64**

Represents the C 64-bit `signed int` datatype. Usually an alias for `c_longlong`.

class ctypes.**c_long**

Represents the C `signed long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_longlong**

Represents the C `signed long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_short**

Represents the C `signed short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_size_t**

Represents the C `size_t` datatype.

class ctypes.**c_ssize_t**

Represents the C `ssize_t` datatype.

New in version 2.7.

class ctypes.**c_ubyte**

Represents the C `unsigned char` datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_uint**

Represents the C `unsigned int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for `c_ulong`.

class ctypes.**c_uint8**

Represents the C 8-bit `unsigned int` datatype. Usually an alias for `c_ubyte`.

class ctypes.**c_uint16**

Represents the C 16-bit unsigned int datatype. Usually an alias for `c_ushort`.

class `ctypes.c_uint32`

Represents the C 32-bit unsigned int datatype. Usually an alias for `c_uint`.

class `ctypes.c_uint64`

Represents the C 64-bit unsigned int datatype. Usually an alias for `c_ulonglong`.

class `ctypes.c_ulong`

Represents the C unsigned long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_ulonglong`

Represents the C unsigned long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_ushort`

Represents the C unsigned short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class `ctypes.c_void_p`

Represents the C void * type. The value is represented as integer. The constructor accepts an optional integer initializer.

class `ctypes.c_wchar`

Represents the C wchar_t datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class `ctypes.c_wchar_p`

Represents the C wchar_t * datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class `ctypes.c_bool`

Represent the C bool datatype (more accurately, _Bool from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

New in version 2.6.

class `ctypes.HRESULT`

Windows only: Represents a HRESULT value, which contains success or error information for a function or method call.

class `ctypes.py_object`

Represents the C PyObject * datatype. Calling this without an argument creates a NULL PyObject * pointer.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `LPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

15.17.2.8. Structured data types

`class ctypes.Union(*args, **kw)`

Abstract base class for unions in native byte order.

`class ctypes.BigEndianStructure(*args, **kw)`

Abstract base class for structures in *big endian* byte order.

`class ctypes.LittleEndianStructure(*args, **kw)`

Abstract base class for structures in *little endian* byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

`class ctypes.Structure(*args, **kw)`

Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a `_fields_` class variable. `ctypes` will create `descriptors` which allow reading and writing the fields by direct attribute accesses. These are the

`_fields_`

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any `ctypes` data type.

For integer type fields like `c_int`, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the `_fields_` class variable *after* the class statement that defines the `Structure` subclass, this allows creating data types that directly or indirectly reference themselves:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                 ]
```

The `_fields_` class variable must, however, be defined before the type is first used (an instance is created, `sizeof()` is called on it, and so on). Later assignments to the `_fields_` class variable will raise an `AttributeError`.

It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the `_fields_` defined in the sub-subclass, if any.

`_pack_`

An optional small integer that allows overriding the alignment of structure fields in the instance. `_pack_` must already be defined when `_fields_` is assigned, otherwise it will have no effect.

`_anonymous_`

An optional sequence that lists the names of unnamed (anonymous) fields. `_anonymous_` must be already defined when `_fields_` is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. `ctypes` will create descriptors in the structure type that allow accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

The `TYPEDESC` structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the `TYPEDESC` instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to defined sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate `_fields_` variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in `_fields_`. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize `_fields_` with the same name, or create new attributes for names not present in `_fields_`.

15.17.2.9. Arrays and pointers

```
class ctypes.Array(*args)
```

Abstract base class for arrays.

The recommended way to create concrete array types is by multiplying any `ctypes` data type with a positive integer. Alternatively, you can subclass this type and define `_length_` and `_type_` class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an `Array`.

`_length_`

A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an `IndexError`. Will be returned by `len()`.

`_type_`

Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

`class ctypes._Pointer`

Private, abstract base class for pointers.

Concrete pointer types are created by calling `POINTER()` with the type that will be pointed to; this is done automatically by `pointer()`.

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so `len()` will raise `TypeError`. Negative subscripts will read from the memory *before* the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

`_type_`

Specifies the type pointed to.

`contents`

Returns the object to which the pointer points. Assigning to this attribute changes the pointer to point to the assigned object.