ctypes

Python 2.x 2.5–2.7 , Python 3.x 3.0–3.6

`ctypes` is a python built-in library that invokes exported functions from native compiled libraries.

Note: Since this library handles compiled code, it is relatively OS dependent.

Examples

Basic ctypes object

The most basic object is an int:

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

Now, `obj` refers to a chunk of memory containing the value 12.

That value can be accessed directly, and even modified:

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

Since `obj` refers to a chunk of memory, we can also find out it's size and location:

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

Basic usage

Let's say we want to use `libc` 's `ntohl` function.

First, we must load `libc.so` :

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

Then, we get the function object:

```
>>> ntohl = libc.ntohl
>>> ntohl
<_FuncPtr object at 0xbaadf00d>
```

And now, we can simply invoke the function:

```
>>> ntohl(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

Which does exactly what we expect it to do.

Common pitfalls

Failing to load a file

The first possible error is failing to load the library. In that case an OSError is usually raised.

This is either because the file doesn't exists (or can't be found by the OS):

```
>>> cdll.LoadLibrary("foobar.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
```

```
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: foobar.so: cannot open shared object file: No such file or directory
```

As you can see, the error is clear and pretty indicative.

The second reason is that the file is found, but is not of the correct format.

```
>>> cdll.LoadLibrary("libc.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: /usr/lib/i386-linux-gnu/libc.so: invalid ELF header
```

In this case, the file is a script file and not a `.so` file. This might also happen when trying to open a `.dll` file on a Linux machine or a 64bit file on a 32bit python interpreter. As you can see, in this case the error is a bit more vague, and requires some digging around.

Failing to access a function

Assuming we successfully loaded the `.so` file, we then need to access our function like we've done on the first example.

When a non-existing function is used, an `AttributeError` is raised:

```
>>> libc.foo
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 360, in __getattr__
    func = self.__getitem__(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 365, in __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /lib/i386-linux-gnu/libc.so.6: undefined symbol: foo
```

Complex usage

Let's combine all of the examples above into one complex scenario: using `libc`'s `lfind` function.

For more details about the function, read the man page . I urge you to read it before going on.

First, we'll define the proper prototypes:

```
>>> compar_proto = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> lfind_proto = CFUNCTYPE(c_void_p, c_void_p, c_void_p, POINTER(c_uint), c_uint, compar_proto)
```

Then, let's create the variables:

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

And now we define the comparison function:

```
>>> def compar(x, y):
        return x.contents.value - y.contents.value
```

Notice that `x` , and `y` are `POINTER(c_int)` , so we need to dereference them and take their values in order to actually compare the value stored in the memory.

Now we can combine everything together:

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

`ptr` is the returned void pointer. If `key` wasn't found in `arr` , the value would be `None` , but in this case we got a valid value.

Now we can convert it and access the value:

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

Also, we can see that `ptr` points to the correct value inside `arr` :

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

---

ctypes arrays

As any good C programmer knows, a single value won't get you that far. What will really get us going are arrays!

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

This is not an actual array, but it's pretty darn close! We created a class that denotes an array of 16 `int` s.

Now all we need to do is to initialize it:

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

Now `arr` is an actual array that contains the numbers from 0 to 15.

They can be accessed just like any list:

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

And just like any other `ctypes` object, it also has a size and a location:

```
>>> sizeof(arr)
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc000l0ff'
```

Wrapping functions for ctypes

In some cases, a C function accepts a function pointer. As avid `ctypes` users, we would like to use those functions, and even pass python function as arguments.

Let's define a function:

```
>>> def max(x, y):
        return x if x >= y else y
```

Now, that function takes two arguments and returns a result of the same type. For the sake of the example, let's assume that type is an int.

Like we did on the array example, we can define an object that denotes that prototype:

```
>>> CFUNCTYPE(c_int, c_int, c_int)
<CFunctionType object at 0xdeadbeef>
```

That prototype denotes a function that returns an `c_int` (the first argument), and accepts two `c_int` arguments (the other arguments).

Now let's wrap the function:

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

Function prototypes have on more usage: They can wrap `ctypes` function (like `libc.ntohl`) and verify that the correct arguments are used when invoking the function.

```
>>> libc.ntohl() # garbage in - garbage out
>>> CFUNCTYPE(c_int, c_int)(libc.ntohl)()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: this function takes at least 1 argument (0 given)
```

Syntax

Parameters

Remarks