# Interfacing Python and C: Advanced “ctypes” Features

<https://dbader.org/blog/python-ctypes-tutorial-part-2>

By [Jim Anderson](https://dbader.org/blog/python-ctypes-tutorial-part-2" \l "author)

Learn advanced patterns for interfacing Python with native libraries, like dealing with C structs from Python and pass-by-value versus pass-by-reference semantics.

The built-in [ctypes module](https://dbader.org/blog/python-ctypes-tutorial) is a powerful feature in Python, allowing you to use existing libraries in other languages by writting simple wrappers in Python itself.

In the [first part](https://dbader.org/blog/python-ctypes-tutorial) of this tutorial, we covered the basics of ctypes. In part two we will dig a little deeper, covering:

* Creating simple Python classes to mirror C structures
* Dealing with C pointers in Python: Pass-by-value vs Pass-by-reference
* Expanding our C structure wrappers to hide complexity from Python code
* Interacting with nested C structures from Python

Again, let’s start by taking a look with the simple C library we will be using and how to build it, and then jump into loading a C library and calling functions in it.

## **Interfacing Python and C: The C Library Testbed**

As with the previous tutorial, all of the code to build and test the examples discussed here (as well as the Markdown for this article) are committed to my [GitHub repository](https://github.com/jima80525/ctypes_example" \t "/home/yg/文档\\x/_blank).

The library consists of two data structures: Point and Line. A Point is a pair of (x, y) coordinates while a Line has a start and end point. There are also a handful of functions which modify each of these types.

Let’s take a closer look at the Point structure and the functions surrounding it. Here’s the corresponding C code split into a Point.h header file and a Point.c implementation:

/\* Point.h \*//\* Simple structure for ctypes example \*/typedef struct {

int x;

int y;} Point;

/\* Point.c \*//\* Display a Point value \*/void show\_point(Point point) {

printf("Point in C is (%d, %d)\n", point.x, point.y);}

/\* Increment a Point which was passed by value \*/void move\_point(Point point) {

show\_point(point);

point.x++;

point.y++;

show\_point(point);}

/\* Increment a Point which was passed by reference \*/void move\_point\_by\_ref(Point \*point) {

show\_point(\*point);

point->x++;

point->y++;

show\_point(\*point);}

/\* Return by value \*/Point get\_point(void) {

static int counter = 0;

Point point = { counter++, counter++ };

printf("Returning Point (%d, %d)\n", point.x, point.y);

return point;}

I won’t go into each of these functions in detail as they are fairly straightforward. The most interesting bit here is the difference between move\_point and move\_point\_by\_ref. We’ll talk a bit later about this when we discuss pass-by-value and pass-by-reference semantics.

We’ll also be using a Line structure, which is composed of two Points:

/\* Line.h \*//\* Compound C structure for our ctypes example \*/typedef struct {

Point start;

Point end;} Line;

/\* Line.c \*/void show\_line(Line line) {

printf("Line in C is (%d, %d)->(%d, %d)\n",

line.start.x, line.start.y,

line.end.x, line.end.y);}

void move\_line\_by\_ref(Line \*line) {

show\_line(\*line);

move\_point\_by\_ref(&line->start);

move\_point\_by\_ref(&line->end);

show\_line(\*line);}

Line get\_line(void) {

Line l = { get\_point(), get\_point() };

return l;}

The Point structure and its associated functions will allow us to show how to wrap structures and deal with memory references in ctypes. The Line structure will allow us to work with nested structures and the complications that arise from that.

The [Makefile](https://github.com/jima80525/ctypes_example/blob/master/tutorial2/Makefile" \t "/home/yg/文档\\x/_blank) in the repo is set up to completely build and run the demo from scratch:

all: point wrappedPoint line

clean:

rm \*.o \*.so

libpoint.so: Point.o

gcc -shared $^ -o $@

libline.so: Point.o Line.o

gcc -shared $^ -o $@

.o: .c

gcc -c -Wall -Werror -fpic $^

point: libpoint.so

./testPoint.py

wrappedPoint: libpoint.so

./testWrappedPoint.py

line: libline.so

./testLine.py

doc:

pandoc ctypes2.md > ctypes2.html

firefox ctypes2.html

To build and run the demo you only need to run the following command in your shell:

$ make

## **Creating Simple Python Classes to Mirror C Structures**

Now that we’ve seen the C code we’ll be using, we can start in on Python and ctypes. We’ll start with a quick wrapper function that will simplify the rest of our code, then we’ll look at how to wrap C structures. Finally, we’ll discuss dealing with C pointers from Python and the differences between pass-by-value and pass-by-reference.

## **Wrapping ctypes Functions**

Before we get into the depths of this tutorial, I’ll show you a utility function we’ll be using throughout. This Python function is called wrap\_function. It takes the object returned from ctypes.CDLL and the name of a function (as a string). It returns a Python object which holds the function and the specified restype and argtypes:

def wrap\_function(lib, funcname, restype, argtypes):

"""Simplify wrapping ctypes functions"""

func = lib.\_\_getattr\_\_(funcname)

func.restype = restype

func.argtypes = argtypes

return func

These are concepts [covered in my previous ctypes tutorial](https://dbader.org/blog/python-ctypes-tutorial), so if this doesn’t make sense, it might be worth reviewing part one again.

## **Mirroring C Structures with Python Classes**

Creating Python classes which mirror C structs requires little code, but does have a little magic behind the scenes:

class Point(ctypes.Structure):

\_fields\_ = [('x', ctypes.c\_int), ('y', ctypes.c\_int)]

def \_\_repr\_\_(self):

return '({0}, {1})'.format(self.x, self.y)

As you can see above, we make use of the \_fields\_ attribute of the class. Please note the single underscore—this is not a [“dunder” function](https://dbader.org/blog/python-dunder-methods). This attribute is a list of tuples and allows ctypes to map attributes from Python back to the underlying C structure.

Let’s look at how it’s used:

>>> libc = ctypes.CDLL('./libpoint.so')>>> show\_point = wrap\_function(libc, 'show\_point', None, [Point])>>> p = Point(1, 2)>>> show\_point(p)'(1, 2)'

Notice that we can access the x and y attributes of the Point class in Python in the \_\_repr\_\_ function. We can also pass the Point directly to the show\_point function in the C library. Ctypes uses the \_fields\_ map to manage the conversions automatically for you. Care should be taken with using the \_fields\_ attribute, however. We’ll look at this in a little more detail in the nested structures section below.

## **Pass-by-value vs Pass-by-reference (pointers)**

In Python we get used to referring to things as either [mutable or immutable](https://www.youtube.com/watch?v=p9ppfvHv2Us" \t "/home/yg/文档\\x/_blank). This controls what happens when you modify an object you’ve passed to a function. For example, number objects are immutable. When you call myfunc in the code below, the value of y does not get modified. The program prints the value 9:

def myfunc(x):

x = x + 2

y = 9myfunc(y)print("this is y", y)

Contrarily, list objects are mutable. In a similar function:

def mylistfunc(x):

x.append("more data")

z = list()mylistfunc(z)print("this is z", z)

As you can see, the list, z, that is passed in to the function is modified and the output is this is z ['more data']

When interfacing with C, we need to take this concept a step further. When we pass a parameter to a function, C always “passes by value”. What this means is that, unless you pass in a pointer to an object, the original object is never changed. Applying this to ctypes, we need to be aware of which values are being passed as pointers and thus need the ctypes.POINTER(Point)type applied to them.

In the example below, we have two versions of the function to move a point: move\_point, which passes by value, and move\_point\_by\_ref which passes by reference.

# --- Pass by value ---print("Pass by value")move\_point = wrap\_function(libc, 'move\_point', None, [Point])a = Point(5, 6)print("Point in Python is", a)move\_point(a)print("Point in Python is", a)print()

# --- Pass by reference ---print("Pass by reference")move\_point\_by\_ref = wrap\_function(libc, 'move\_point\_by\_ref', None,

[ctypes.POINTER(Point)])a = Point(5, 6)print("Point in Python is", a)move\_point\_by\_ref(a)print("Point in Python is", a)print()

The output from these two code sections looks like this:

Pass by value

Point in Python is (5, 6)

Point in C is (5, 6)

Point in C is (6, 7)

Point in Python is (5, 6)

Pass by reference

Point in Python is (5, 6)

Point in C is (5, 6)

Point in C is (6, 7)

Point in Python is (6, 7)

As you can see, when we call move\_point, the C code can change the value of the Point, but that change is not reflected in the Python object. When we call move\_point\_by\_ref, however, the change is visible in the Python object. This is because we passed the address of the memory which holds that value and the C code took special care (via using the ->accessor) to modify that memory.

When working in cross-language interfaces, memory access and memory management are important aspects to keep in mind.

## **Accessing C Structs from Python – An OOP Wrapper**

We saw above that providing a simple wrapper to a C structure is quite easy using ctypes. We can also expand this wrapper to make it behave like a “proper” Python class instead of a C struct using object-oriented programming principles.

Here’s an example:

class Point(ctypes.Structure):

\_fields\_ = [('x', ctypes.c\_int), ('y', ctypes.c\_int)]

def \_\_init\_\_(self, lib, x=None, y=None):

if x:

self.x = x

self.y = y

else:

get\_point = wrap\_function(lib, 'get\_point', Point, None)

self = get\_point()

self.show\_point\_func = wrap\_function(lib, 'show\_point', None, [Point])

self.move\_point\_func = wrap\_function(lib, 'move\_point', None, [Point])

self.move\_point\_ref\_func = wrap\_function(lib, 'move\_point\_by\_ref', None,

[ctypes.POINTER(Point)])

def \_\_repr\_\_(self):

return '({0}, {1})'.format(self.x, self.y)

def show\_point(self):

self.show\_point\_func(self)

def move\_point(self):

self.move\_point\_func(self)

def move\_point\_by\_ref(self):

self.move\_point\_ref\_func(self)

You’ll see the \_fields\_ and \_\_repr\_\_ attributes are the same as we had in our simple wrapper, but now we’ve added a constructor and wrapping functions for each method we’ll use.

The interesting code is all in the constructor. The initial part initializes the xand y fields. You can see that we have two methods to achieve this. If the user passed in values, we can directly assign those to the fields. If the default values were used, we call the get\_point function in the library and assign that directly to self.

Once we’ve initialized the fields in our Point class, we then wrap the functions into attributes of our class to allow them to be accessed in a more object oriented manner.

In the testWrappedPoint module, we do the same tests we did with our Point class but instead of passing the Point class to the function, move\_point\_by\_ref(a), we call the function on the object a.move\_point\_by\_ref().

## **Accessing Nested C Structures From Python**

Finally, we’re going to look at how to use nested structures in ctypes. The obvious next step in our example is to extend a Point to a Line:

class Line(ctypes.Structure):

\_fields\_ = [('start', testPoint.Point), ('end', testPoint.Point)]

def \_\_init\_\_(self, lib):

get\_line = wrap\_function(lib, 'get\_line', Line, None)

line = get\_line()

self.start = line.start

self.end = line.end

self.show\_line\_func = wrap\_function(lib, 'show\_line', None, [Line])

self.move\_line\_func = wrap\_function(lib, 'move\_line\_by\_ref', None,

[ctypes.POINTER(Line)])

def \_\_repr\_\_(self):

return '{0}->{1}'.format(self.start, self.end)

def show\_line(self):

self.show\_line\_func(self)

def moveLine(self):

self.move\_line\_func(self)

Most of this class should look fairly familiar if you’ve been following along. The one interesting difference is how we initialize the \_fields\_ attribute. You’ll remember in the Point class we could assign the returned value from get\_point() directly to self. This doesn’t work with our Line wrapper as the entries in the \_fields\_ list are not basic CTypes types, but rather a subclass of one of them. Assigning these directly tends to mess up how the value is stored so that the Python attributes you add to the class are inaccessible.

The basic rule I’ve found in wrapping structures like this is to only add the Python class attributes at the top level and leave the inner structures (i.e. Point) with the simple \_fields\_ attribute.

## **Advanced ctypes Features – Conclusion**

In this tutorial we covered some more advanced topics in using the ctypesmodule to interface Python with external C libraries. I found several resources out there while researching:

The [ctypesgen project](https://github.com/davidjamesca/ctypesgen" \t "/home/yg/文档\\x/_blank) has tools which will auto generate Python wrapping modules for C header files. I spent some time playing with this and it looks quite good.

The idea for the wrap\_function function was lifted shamelessly from some ctypes tips [here](https://www.cs.unc.edu/~gb/blog/2007/02/11/ctypes-tricks/" \t "/home/yg/文档\\x/_blank).

In the [first part of this tutorial](https://dbader.org/blog/python-ctypes-tutorial), we covered the basics of ctypes, so be sure to check there if you’re looking for a ctypes primer. And, finally, if you’d like to see and play with the code I wrote while working on this, please visit my [GitHub repository](https://github.com/jima80525/ctypes_example" \t "/home/yg/文档\\x/_blank). This tutorial is in the tutorial2 directory.