Extending Python With C Libraries and the “ctypes” Module

(https://facebook.com/sharer/sharer.php?u=https://dbader.org/blog/python-ctypes-tutorial)

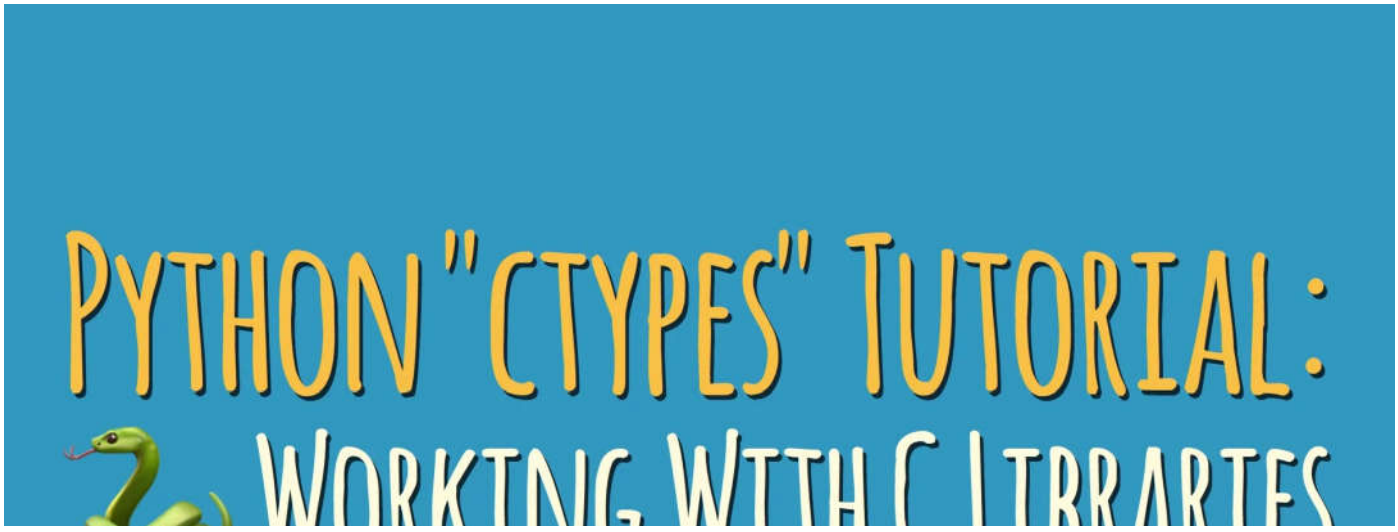
(https://twitter.com/intent/tweet/?text=Extending%20Python%20With%20C%20Libraries%20and%20the%20%E2%80%9Cctypes%E2%80%9D%20Module&url=https://dbader.org/blog/python-ctypes-tutorial)

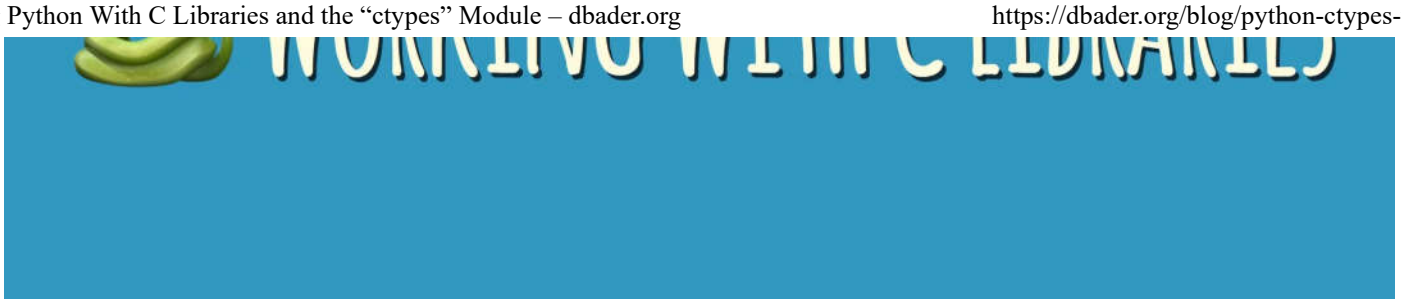
(https://plus.google.com/share?url=https://dbader.org/blog/python-ctypes-tutorial)

(https://www.linkedin.com/shareArticle?mini=true&url=https://dbader.org/blog/python-ctypes-tutorial&title=Extending%20Python%20With%20C%20Libraries%20and%20the%20%E2%80%9Cctypes%E2%80%9D%20Module&summary=Extending%20Python%20With%20C%20Libraries%20and%20the%20%E2%80%9Cctypes%E2%80%9D%20Module&source=https://dbader.org/blog/python-ctypes-tutorial)

By Jim Anderson

An end-to-end tutorial of how to extend your Python programs with libraries written in C, using the built-in “ctypes” module.





The built-in `ctypes` module (https://docs.python.org/3/library/ctypes.html) is a powerful feature in Python, allowing you to use existing libraries in other languages by writting simple wrappers in Python itself.

Unfortunately it can be a bit tricky to use. In this article we'll explore some of the basics of `ctypes`. We'll cover:

- Loading C libraries
- Calling a simple C function
- Passing mutable and immutable strings
- Managing memory

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.

A Simple C Library That Can Be Used From Python

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).

I'll walk through a little bit about the C library before we get into `ctypes`.

The C code we'll use in this tutorial is designed to be as simple as possible while demonstrating the concepts we're covering. It's more of a “toy example” and not intended to be useful on its own. Here are the functions we'll be using:

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

The `simple_function` function simply returns counting numbers. Each time it is called in increments `counter` and returns that value.

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

The `add_one_to_string` function adds one to each character in a char array that is passed in. We'll use this to talk about Python's immutable strings and how to work around them when we need to.

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

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

This pair of functions allocate and free a string in the C context. This will provide the framework for talking about memory management in `ctypes`.

Finally, we need a way to build this source file into a library. While there are many tools I prefer to use `make` (https://en.wikipedia.org/wiki/Make_(software)), I use it for projects like this because of its low overhead and ubiquity. Make is available on all Linux-like systems.

Here's a snippet from the Makefile (https://en.wikipedia.org/wiki/Makefile) which builds the C library into a `.so` file:

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

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

The Makefile in the repo is set up to completely build and run the demo from scratch; you only need to run the following command in your shell:

```
$ make
```

Loading a C Library With Python’s “ctypes” Module

Ctypes allows your to load a shared library (https://www.dwheeler.com/program-library/Program-Library-HOWTO/x36.html) (“DLL” on Windows) and access methods directly from it, provided you take care to “marshal” (https://en.wikipedia.org/wiki/Marshalling_(computer_science)) the data properly.

The most basic form of this is:

```
import ctypes

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

Note that this assumes that your shared library is in the same directory as your script and that you are calling the script from that directory. There are a lot of OS-specific details around library search paths which are beyond the scope of this article, but if you can package the `.py` file alongside the shared library, you can use something like this:

```
libname = os.path.abspath(
    os.path.join(os.path.dirname(__file__), "libclib1.so"))

libc = ctypes.CDLL(libname)
```

This will allow you to call the script from any directory.

Once you have loaded the library, it is stored in a Python object which has methods for each exported function.

Calling Simple Functions with ctypes

The great thing about `ctypes` is that it makes the simple things quite simple. Simply calling a function with no parameters is trivial. Once you have loaded the library, the function is just a method of the library object.

```
import ctypes

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

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

You’ll remember that the C function we’re calling returns counting numbers as `int` objects.

Again, ctypes makes easy things easy—passing ints around works seamlessly and does pretty much what you expect it to.

Dealing with Mutable and Immutable Strings as ctypes Parameters

While basic types, ints and floats, generally get marshalled by ctypes trivially, strings pose a problem. In Python, strings are *immutable*, meaning they cannot change. (https://www.youtube.com/watch?v=p9ppfvHv2Us) This produces some odd behavior when passing strings in ctypes.

For this example we'll use the add_one_to_string function shown in the C library above. If we call this passing in a Python string it runs, but does not modify the string as we might expect. This Python code:

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

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

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

Results in this output:

```
Calling C function which tries to modify Python string
Before: starting string
After:  starting string
```

After some testing, I proved to myself that the original_string is not available in the C function at all when doing this. The original string was unchanged, mainly because the C function modified some other memory, not the string. So, not only does the C function not do what you want, but it also modifies memory that it should not, leading to potential memory corruption issues.

If we want the C function to have access to the string we need to do a little marshalling work up front. Fortunately, ctypes makes this fairly easy, too.

We need to convert the original string to bytes using str.encode, and then pass this to the constructor for a ctypes.string_buffer. String_buffers *are* mutable, and they are passed to C as a char * as you would expect.

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

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

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

Running this code prints:

```
Calling C function with mutable buffer this time
Before: b'starting string'
After:  b'tubsujoh!tusjoh'
```

Note that the `string_buffer` is printed as a byte array on the Python side.

Specifying Function Signatures in ctypes

Before we get to the final example for this tutorial, we need to take a brief aside and talk about how `ctypes` passes parameters and returns values. As we saw above we can specify the return type if needed.

We can do a similar specifcation of the function parameters. Ctypes will figure out the type of the pointer and create a default mapping to a Python type, but that is not always what you want to do. Also, providing a function signature allows Python to check that you are passing in the correct parameters when you call a C function, otherwise crazy things can happen.

Because each of the functions in the loaded library is actually a Python object which has its own properties, specifying the return value is quite simple. To specify the return type of a function, you get the function object and set the `restype` property like this:

```
alloc_func = libc.alloc_C_string
alloc_func.restype = ctypes.POINTER(ctypes.c_char)
```

Similarly, you can specify the types of any arguments passed in to the C function by setting the argtypes property to a list of types:

```
free_func = libc.free_C_string
free_func.argtypes = [ctypes.POINTER(ctypes.c_char), ]
```

I’ve found several different clever methods in my studies for how to simplify specifying these, but in the end they all come down to these properties.

Memory Management Basics in ctypes

One of the great features of moving from C to Python is that you no longer need to spend time doing manual memory management. The golden rule when doing `ctypes`, or any cross-language marshalling is that *the language that allocates the memory also needs to free the memory*.

In the example above this worked quite well as Python allocated the string buffers we were passing around so it could then free that memory when it was no longer needed.

Frequently, however, the need arises to allocate memory in C and then pass it to Python for some manipulation. This works, but you need to take a few more steps to ensure you can pass the memory pointer back to C so it can free it when we’re done.

For this example, I’ll use these two C functions, `alloc_C_string` and `free_C_string`. In the example code both functions print out the memory pointer they are manipulating to make it clear what is happening.

As mentioned above, we need to be able to keep the actual pointer to the memory that `alloc_C_string` allocated so that we can pass it back to `free_C_string`. To do this, we need to tell ctype that `alloc_C_string` should return a `ctypes.POINTER` to a `ctypes.c_char`. We saw that earlier.

The `ctypes.POINTER` objects are not overly useful, but they can be converted to objects which are useful. Once we convert our string to a `ctypes.c_char`, we can access it’s value attribute to get the bytes in Python.

Putting that all together looks like this:

```
alloc_func = libc.alloc_C_string

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

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

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

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

Once we’ve used the data we allocated in C, we need to free it. The process is quite similar, specifying the `argtypes` attribute instead of `restype`:

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

Python’s “ctypes” Module – Conclusion

Python’s built-in `ctypes` feature allows you to interact with C code from Python quite easily, using a few basic rules to allow you to specify and call those functions. However, you must be careful about memory management and ownership.

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).

Also, be sure to check out part two of this tutorial (/blog/python-ctypes-tutorial-part-2) where you’ll learn more about advanced features and patterns in using the `ctypes` library to interface Python with C code.

This article was filed under: ctypes (/blog/tags/ctypes), optimization (/blog/tags/optimization), programming (/blog/tags/programming), and python (/blog/tags/python).

Related Articles:

- Interfacing Python and C: Advanced “ctypes” Features (/blog/python-ctypes-tutorial-part-2) – 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.
- Interfacing Python and C: The CFFI Module (/blog/python-cffi) – How to use Python’s built-