Python Bindings: Calling C or C++ From Python

Python Bindings: Calling C or C++ From Python

- Python Bindings Overview
  - Marshalling Data Types
  - Understanding Mutable and Immutable Values
  - Managing Memory
  - Setting Up Your Environment
  - Using the `invoke` Tool
  - C or C++ Source
- `ctypes`
  - How It's Installed
  - Calling the Function
    - Library Loading
    - Calling Your Function
  - Strengths and Weaknesses
- `CFFI`
  - How It's Installed
  - Calling the Function
    - Write the Bindings
    - Build the Python Bindings
    - Calling Your Function
  - Strengths and Weaknesses
- `PyBind11`
  - How It's Installed
  - Calling the Function
    - Writing the Bindings
    - Build the Python Bindings
    - Calling Your Function
  - Strengths and Weaknesses
- `Cython`
  - How It's Installed
  - Calling the Function
    - Write the Bindings
    - Build the Python Bindings
    - Calling Your Function
  - Strengths and Weaknesses
- Other Solutions
  - `PyBindGen`
  - `Boost.Python`
  - `SIP`
  - `Cppyy`
  - `Shiboken`
  - `SWIG`
- Conclusion

Are you a Python developer with a C or C++ library you'd like to use from Python? If so, then **Python bindings** allow you to call functions and pass data from Python to C or C++, letting you take advantage of the strengths of both languages. Throughout this tutorial, you'll see an overview of some of the tools you can use to create Python bindings.

In this tutorial, you'll learn about:

- Why you want to **call C or C++** from Python
- How to **pass data** between C and Python
- What **tools and methods** can help you create Python bindings

This tutorial is aimed at intermediate Python developers. It assumes basic knowledge of Python and some understanding of functions and data types in C or C++. You can get all of the example code you'll see in this tutorial by clicking on the link below:

Let's dive into looking at Python bindings!

Python Bindings Overview

Before you dive into *how* to call C from Python, it's good to spend some time on *why*. There are several situations where creating Python bindings to call a C library is a great idea:

1. **You already have a large, tested, stable library written in C++** that you'd like to take advantage of in Python. This may be a communication library or a library to talk to a specific piece of hardware. What it does is unimportant.
2. **You want to speed up a particular section of your Python code** by converting a critical section to C. Not only does C have faster execution speed, but it also allows you to break free from the limitations of the GIL, provided you're careful.
3. **You want to use Python test tools** to do large-scale testing of their systems.

All of the above are great reasons to learn to create Python bindings to interface with your C library.

Note: Throughout this tutorial, you'll be creating Python bindings to both C *and* C++. Most of the general concepts apply to both languages, and so C will be used unless there's a specific difference between the two languages. In general, each of the tools will support either C *or* C++, but not both.

Let's get started!

Marshalling Data Types

Wait! Before you start writing Python bindings, take a look at how Python and C store data and what types of issues this will cause. First, let's define **marshalling**. This concept is defined by Wikipedia as follows:

> The process of transforming the memory representation of an object to a data format suitable for storage or transmission. (Source)

For your purposes, marshalling is what the Python bindings are doing when they prepare data to move it from Python to C or vice versa. Python bindings need to do marshalling because Python and C store data in different ways. C stores data in the most compact form in memory possible. If you use an `uint8_t`, then it will only use 8 bits of memory total.

In Python, on the other hand, everything is an **object**. This means that each integer uses several bytes in memory. How many will depend on which version of Python you're running, your operating system, and other factors. This means that your Python bindings will need to convert a **C integer** to a **Python integer** for each integer passed across the boundary.

Other data types have similar relationships between the two languages. Let's look at each in turn:

- **Integers** store counting numbers. Python stores integers with arbitrary precision, meaning that you can store very, very, large numbers. C specifies the exact sizes of integers. You need to be aware of data sizes when you're moving between languages to prevent Python integer values from overflowing C integer variables.
- **Floating-point numbers** are numbers with a decimal place. Python can store much larger (and much smaller) floating-point numbers than C. This means that you'll also have to pay attention to those values to ensure they stay in range.
- **Complex numbers** are numbers with an imaginary part. While Python has built-in complex numbers, and C has complex numbers, there's no built-in method for marshalling between them. To marshal complex numbers, you'll need to build a `struct` or `class` in the C code to manage them.
- **Strings** are sequences of characters. For being such a common data type, strings will prove to be rather tricky when you're creating Python bindings. As with other data types, Python and C store strings in quite different formats. (Unlike the other data types, this is an area where C and C++ differ as well, which adds to the fun!) Each of the solutions you'll examine have slightly different methods for dealing with strings.
- **Boolean variables** can have only two values. Since they're supported in C, marshalling them will prove to be fairly straightforward.

Besides data type conversions, there are other issues you'll need to think about as you build your Python bindings. Let's keep exploring them.

Understanding Mutable and Immutable Values

In addition to all of these data types, you'll also have to be aware of how Python objects can be **mutable** or **immutable**. C has a similar concept with function parameters when talking about **pass-by-value** or **pass-by-reference**. In C, *all* parameters are pass-by-value. If you want to allow a function to change a variable in the caller, then you need to pass a pointer to that variable.

You might be wondering if you can get around the **immutable restriction** by simply passing an immutable object to C using a pointer. Unless you go to ugly and non-portable extremes, Python won't give you a pointer to an object, so this just doesn't work. If you want to modify a Python object in C, then you'll need to take extra steps to achieve this. These steps will be dependent on which tools you use, as you'll see below.

So, you can add immutability to your checklist of items to consider as you create Python bindings. Your final stop on the grand tour of creating this checklist is how to handle the different ways in which Python and C deal with memory management.

Managing Memory

C and Python **manage memory** differently. In C, the developer must manage all memory allocations and ensure they're freed once and only once. Python takes care of this for you using a garbage collector.

While each of these approaches has its advantages, it does add an extra wrinkle into creating Python bindings. You'll need to be aware of **where the memory for each object was allocated** and ensure that it's only freed on the same side of the language barrier.

For example, a Python object is created when you set `x = 3`. The memory for this is allocated on the Python side and needs to be garbage collected. Fortunately, with Python objects, it's quite difficult to do anything else. Take a look at the converse in C, where you directly allocate a block of memory:

```
int* iPtr = (int*)malloc(sizeof(int));
```

When you do this, you need to ensure that this pointer is freed in C. This may mean manually adding code to your Python bindings to do this.

That rounds out your checklist of general topics. Let's start setting up your system so you can write some code!

Setting Up Your Environment

For this tutorial, you're going to be using pre-existing C and C++ libraries from the Real Python GitHub repo to show a test of each tool. The intent is that you'll be able to use these ideas for any C library. To follow along with all of the examples here, you'll need to have the following:

- A **C++ library** installed and knowledge of the path for command-line invocation
- Python development tools:
  - For Linux, this is the `python3-dev` or `python3-devel` package, depending on your distro.
  - For Windows, there are multiple options.
- **Python 3.6** or greater
- A **virtual environment** (recommended, but not required)
- The `invoke` tool

The last one might be new to you, so let's take a closer look at it.

Using the `invoke` Tool

`invoke` is the tool you'll be using to build and test your Python bindings in this tutorial. It has a similar purpose to `make` but uses Python instead of Makefiles. You'll need to install `invoke` in your virtual environment using `pip`:

```
$ python3 -m pip install invoke
```

To run it, you type `invoke` followed by the task you wish to execute:

```
$ invoke build-cmult
==================================================
= Building C Library
* Complete
```

To see which tasks are available, you use the `--list` option:

```
$ invoke --list
Available tasks:

  all              Build and run all tests
  build-cffi       Build the CFFI Python bindings
  build-cmult      Build the shared library for the sample C code
  build-cppmult    Build the shared library for the sample C++ code
  build-cython     Build the cython extension module
```

```
  build-pybind11   Build the pybind11 wrapper library
  clean            Remove any built objects
  test-cffi        Run the script to test CFFI
  test-ctypes      Run the script to test ctypes
  test-cython      Run the script to test Cython
  test-pybind11    Run the script to test PyBind11
```

Note that when you look in the `tasks.py` file where the `invoke` tasks are defined, you'll see that the name of the second task listed is `build_cffi`. However, the output from `--list` shows it as `build-cffi`. The minus sign (`-`) can't be used as part of a Python name, so the file uses an underscore (`_`) instead.

For each of the tools you'll examine, there will be a `build-` and a `test-` task defined. For example, to run the code for `CFFI`, you could type `invoke build-cffi test-cffi`. An exception is `ctypes`, as there's no build phase for `ctypes`. In addition, there are two special tasks added for convenience:

- `invoke all` runs the build and test tasks for all tools.
- `invoke clean` removes any generated files.

Now that you've got a feeling for how to run the code, let's take a peek at the C code you'll be wrapping before hitting the tools overview.

C or C++ Source

In each of the example sections below, you'll be **creating Python bindings** for the same function in either C or C++. These sections are intended to give you a taste of what each method looks like, rather than an in-depth tutorial on that tool, so the function you'll wrap is small. The function you'll create Python bindings for takes an `int` and a `float` as input parameters and returns a `float` that's the product of the two numbers:

```
// cmult.c
float cmult(int int_param, float float_param) {
    float return_value = int_param * float_param;
    printf("    In cmult : int: %d float %.1f returning  %.1f\n", int_param,
            float_param, return_value);
    return return_value;
}
```

The C and C++ functions are almost identical, with minor name and string differences between them. You can get a copy of all of the code by clicking on the link below:

Get Sample Code: Click here to get the sample code you'll use to learn about Python Bindings in this tutorial.

Now you've got the repo cloned and your tools installed, you can build and test the tools. So let's dive into each section below!

`ctypes`

You'll start with `ctypes`, which is a tool in the standard library for creating Python bindings. It provides a low-level toolset for loading shared libraries and marshalling data between Python and C.

How It's Installed

One of the big advantages of `ctypes` is that it's part of the Python standard library. It was added in Python version 2.5, so it's quite likely you already have it. You can `import` it just like you do with the `sys` or `time` modules.

Calling the Function

All of the code to load your C library and call the function will be in your Python program. This is great since there are no extra steps in your process. You just run your program, and everything is taken care of. To create your Python bindings in `ctypes`, you need to do these steps:

1. **Load** your library.
2. **Wrap** some of your input parameters.
3. **Tell** `ctypes` the return type of your function.

You'll look at each of these in turn.

Library Loading

`ctypes` provides several ways for you to load a shared library, some of which are platform-specific. For your example, you'll create a `ctypes.CDLL` object directly by passing in the full path to the shared library you want:

```
# ctypes_test.py
import ctypes
import pathlib

if __name__ == "__main__":
    # Load the shared library into ctypes
    libname = pathlib.Path().absolute() / "libcmult.so"
    c_lib = ctypes.CDLL(libname)
```

This will work for cases when the shared library is in the same directory as your Python script, but be careful when you attempt to load libraries that are from packages other than your Python bindings. There are many details for loading libraries and finding paths in the `ctypes` documentation that are platform and situation-specific.

NOTE: Many platform-specific issues can arise during library loading. It's best to make incremental changes once you get an example working.

Now that you have the library loaded into Python, you can try calling it!

Calling Your Function

Remember that the function prototype for your C function is as follows:

```
// cmult.h
float cmult(int int_param, float float_param);
```

You need to pass in an integer and a float and can expect to get a float returned. Integers and floats have native support in both Python and in C, so you expect this case will work for reasonable values.

Once you've loaded the library into your Python bindings, the function will be an attribute of `c_lib`, which is the `CDLL` object you created earlier. You can try to call it just like this:

```
x, y = 6, 2.3
answer = c_lib.cmult(x, y)
```

Oops! This doesn't work. This line is commented out in the example repo because it fails. If you attempt to run with that call, then Python will complain with an error:

```
$ invoke test-ctypes
Traceback (most recent call last):
  File "ctypes_test.py", line 16, in <module>
    answer = c_lib.cmult(x, y)
ctypes.ArgumentError: argument 2: <class 'TypeError'>: Don't know how to convert
parameter 2
```

It looks like you need to tell `ctypes` about any parameters that aren't integers. `ctypes` doesn't have any knowledge of the function unless you tell it explicitly. Any parameter that's not marked otherwise is assumed to be an integer. `ctypes` doesn't know how to convert the value `2.3` that's stored in `y` to an integer, so it fails.

To fix this, you'll need to create a `c_float` from the number. You can do that in the line where you're calling the function:

```
# ctypes_test.py
answer = c_lib.cmult(x, ctypes.c_float(y))
print(f"    In Python: int: {x} float {y:.1f} return val {answer:.1f}")
```

Now, when you run this code, it returns the product of the two numbers you passed in:

```
$ invoke test-ctypes
    In cmult : int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 48.0
```

Wait a minute... `6` multiplied by `2.3` is not `48.0`!

It turns out that, much like the input parameters, `ctypes` **assumes** your function returns an `int`. In actuality, your function returns a `float`, which is getting marshalled incorrectly. Just like the input parameter, you need to tell `ctypes` to use a different type. The syntax here is slightly different:

```
# ctypes_test.py
c_lib.cmult.restype = ctypes.c_float
answer = c_lib.cmult(x, ctypes.c_float(y))
print(f"    In Python: int: {x} float {y:.1f} return val {answer:.1f}")
```

That should do the trick. Let's run the entire `test-ctypes` target and see what you've got. Remember, the first section of output is **before** you fixed the `restype` of the function to be a float:

```
$ invoke test-ctypes
================================================
= Building C Library
* Complete
================================================
= Testing ctypes Module
    In cmult : int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 48.0

    In cmult : int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 13.8
```

That's better! While the first, uncorrected version is returning the wrong value, your fixed version agrees with the C function. Both C and Python get the same result! Now that it's working, take a look at why you may or may not want to use `ctypes`.

Strengths and Weaknesses

The biggest advantage `ctypes` has over the other tools you'll examine here is that it's **built into the standard library**. It also requires no extra steps, as all of the work is done as part of your Python program.

In addition, the concepts used are low-level, which makes exercises like the one you just did manageable. However, more complex tasks grow cumbersome with the lack of automation. In the next section, you'll see a tool that adds some automation to the process.

`CFFI`

`CFFI` is the **C Foreign Function Interface** for Python. It takes a more automated approach to generate Python bindings. `CFFI` has multiple ways in which you can build and use your Python bindings. There are two different options to select from, which gives you four possible modes:

- **ABI vs API:** API mode uses a C compiler to generate a full Python module, whereas ABI mode loads the shared library and interacts with it directly. Without running the compiler, getting the structures and parameters correct is error-prone. The documentation heavily recommends using the API mode.
- **in-line vs out-of-line:** The difference between these two modes is a trade-off between speed and convenience:
    - **In-line mode** compiles the Python bindings every time your script runs. This is convenient, as you don't need an extra build step. It does, however, slow down your program.
    - **Out-of-line mode** requires an extra step to generate the Python bindings a single time and then uses them each time the program is run. This is much faster, but that may not matter for your application.

For this example, you'll use the API out-of-line mode, which produces the fastest code and, in general, looks similar to other Python bindings you'll create later in this tutorial.

How It's Installed

Since `CFFI` is not a part of the standard library, you'll need to install it on your machine. It's recommended that you create a virtual environment for this. Fortunately, `CFFI` installs with `pip`:

```
$ python3 -m pip install cffi
```

This will install the package into your virtual environment. If you've already installed from the `requirements.txt`, then this should be taken care of. You can take a look at `requirements.txt` by accessing the repo at the link below:

Get Sample Code: Click here to get the sample code you'll use to learn about Python Bindings in this tutorial.

Now that you have `CFFI` installed, it's time to take it for a spin!

Calling the Function

Unlike `ctypes`, with `CFFI` you're creating a full Python module. You'll be able to `import` the module like any other module in the standard library. There is some extra work you'll have to do to build your Python module. To use your `CFFI` Python bindings, you'll need to take the following steps:

- **Write** some Python code describing the bindings.
- **Run** that code to generate a loadable module.
- **Modify** the calling code to import and use your newly created module.

That might seem like a lot of work, but you'll walk through each of these steps and see how it works.

Write the Bindings

`CFFI` provides methods to read a **C header file** to do most of the work when generating Python bindings. In the documentation for `CFFI`, the code to do this is placed in a separate Python file. For this example, you'll place that code directly into the build tool `invoke`, which uses Python files as input. To use `CFFI`, you start by creating a `cffi.FFI` object, which provides the three methods you need:

```
# tasks.py
import cffi
...
""" Build the CFFI Python bindings """
print_banner("Building CFFI Module")
ffi = cffi.FFI()
```

Once you have the FFI, you'll use `.cdef()` to process the contents of the header file automatically. This creates wrapper functions for you to marshal data from Python:

```
# tasks.py
this_dir = pathlib.Path().absolute()
h_file_name = this_dir / "cmult.h"
with open(h_file_name) as h_file:
    ffi.cdef(h_file.read())
```

Reading and processing the header file is the first step. After that, you need to use `.set_source()` to describe the source file that `CFFI` will generate:

```
# tasks.py
ffi.set_source(
    "cffi_example",
    # Since you're calling a fully-built library directly, no custom source
    # is necessary. You need to include the .h files, though, because behind
    # the scenes cffi generates a .c file that contains a Python-friendly
    # wrapper around each of the functions.
    '#include "cmult.h"',
    # The important thing is to include the pre-built lib in the list of
    # libraries you're linking against:
    libraries=["cmult"],
    library_dirs=[this_dir.as_posix()],
    extra_link_args=["-Wl,-rpath,."],
)
```

Here's a breakdown of the parameters you're passing in:

- `"cffi_example"` is the base name for the source file that will be created on your file system. `CFFI` will generate a `.c` file, compile it to a `.o` file, and link it to a `.<system-description>.so` or `.<system-description>.dll` file.
- `'#include "cmult.h"'` is the custom C source code that will be included in the generated source before it's compiled. Here, you just include the `.h` file for which you're generating bindings, but this can be used for some interesting customizations.
- `libraries=["cmult"]` tells the linker the name of your pre-existing C library. This is a list, so you can specify several libraries if required.
- `library_dirs=[this_dir.as_posix(),]` is a list of directories that tells the linker where to look for the above list of libraries.
- `extra_link_args=['-Wl,-rpath,.']` is a set of options that generate a shared object, which will look in the current path (`.`) for other libraries it needs to load.

Build the Python Bindings

Calling `.set_source()` doesn't build the Python bindings. It only sets up the metadata to describe what will be generated. To build the Python bindings, you need to call `.compile()`:

```
# tasks.py
ffi.compile()
```

This wraps things up by generating the `.c` file, `.o` file, and the shared library. The `invoke` task you just walked through can be run on the command line to build the Python bindings:

```
$ invoke build-cffi
====================================================
= Building C Library
* Complete
====================================================
= Building CFFI Module
* Complete
```

You have your `CFFI` Python bindings, so it's time to run this code!

Calling Your Function

After all of the work you did to configure and run the `CFFI` compiler, using the generated Python bindings looks just like using any other Python module:

```
# cffi_test.py
import cffi_example

if __name__ == "__main__":
    # Sample data for your call
    x, y = 6, 2.3

    answer = cffi_example.lib.cmult(x, y)
    print(f"    In Python: int: {x} float {y:.1f} return val {answer:.1f}")
```

You import the new module, and then you can call `cmult()` directly. To test it out, use the `test-cffi` task:

```
$ invoke test-cffi
==================================================
= Testing CFFI Module
    In cmult : int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 13.8
```

This runs your `cffi_test.py` program, which tests out the new Python bindings you've created with `CFFI`. That completes the section on writing and using your `CFFI` Python bindings.

Strengths and Weaknesses

It might seem that `ctypes` requires less work than the `CFFI` example you just saw. While this is true for this use case, `CFFI` scales to larger projects much better than `ctypes` due to **automation** of much of the function wrapping.

`CFFI` also produces quite a different user experience. `ctypes` allows you to load a pre-existing C library directly into your Python program. `CFFI`, on the other hand, creates a new Python module that can be loaded like other Python modules.

What's more, with the **out-of-line-API** method you used above, the time penalty for creating the Python bindings is done once when you build it and doesn't happen each time you run your code. For small programs, this might not be a big deal, but `CFFI` scales better to larger projects in this way, as well.

Like `ctypes`, using `CFFI` only allows you to interface with C libraries directly. C++ libraries require a good deal of work to use. In the next section, you'll see a Python bindings tool that focuses on C++.

`PyBind11`

`PyBind11` takes a quite different approach to create Python bindings. In addition to shifting the focus from C to C++, it also **uses C++ to specify and build the module**, allowing it to take advantage of the metaprogramming tools in C++. Like `CFFI`, the Python bindings generated from `PyBind11` are a full Python module that can be imported and used directly.

`PyBind11` is modeled after the `Boost::Python` library and has a similar interface. It restricts its use to C++11 and newer, however, which allows it to simplify and speed things up compared to Boost, which supports everything.

How It's Installed

The First Steps section of the `PyBind11` documentation walks you through how to download and build the test cases for `PyBind11`. While this doesn't appear to be strictly required, working through these steps will ensure you've got the proper C++ and Python tools set up.

Note: Most of the examples for `PyBind11` use `cmake`, which is a fine tool for building C and C++ projects. For this demo, however, you'll continue to use the `invoke` tool, which follows the instructions in the Building Manually section of the docs.

You'll want to install this tool into your virtual environment:

```
$ python3 -m pip install pybind11
```

`PyBind11` is an all-header library, similar to much of Boost. This allows `pip` to install the actual C++ source for the library directly into your virtual environment.

Calling the Function

Before you dive in, please note that **you're using a different C++ source file**, `cppmult.cpp`, instead of the C file you used for the previous examples. The function is essentially the same in both languages.

Writing the Bindings

Similar to `CFFI`, you need to create some code to tell the tool how to build your Python bindings. Unlike `CFFI`, this code will be in C++ instead of Python. Fortunately, there's a minimal amount of code required:

```
// pybind11_wrapper.cpp
#include <pybind11/pybind11.h>
#include <cppmult.hpp>

PYBIND11_MODULE(pybind11_example, m) {
    m.doc() = "pybind11 example plugin"; // Optional module docstring
    m.def("cpp_function", &cppmult, "A function that multiplies two numbers");
}
```

Let's look at this a piece at a time, as `PyBind11` packs a lot of information into a few lines.

The first two lines include the `pybind11.h` file and the header file for your C++ library, `cppmult.hpp`. After that, you have the `PYBIND11_MODULE` macro. This expands into a block of C++ code that's well described in the `PyBind11` source:

> This macro creates the entry point that will be invoked when the Python interpreter imports an extension module. The module name is given as the first argument and it should not be in quotes. The second macro argument defines a variable of type `py::module` which can be used to initialize the module. (Source)

What this means for you is that, for this example, you're creating a module called `pybind11_example` and that the rest of the code will use `m` as the name of the `py::module` object. On the next line, inside the C++ function you're defining, you create a docstring for the module. While this is optional, it's a nice touch to make your module more Pythonic.

Finally, you have the `m.def()` call. This will define a function that's exported by your new Python bindings, meaning it will be visible from Python. In this example, you're passing three parameters:

- `cpp_function` is the exported name of the function that you'll use in Python. As this example shows, it doesn't need to match the name of the C++ function.
- `&cppmult` takes the address of the function to be exported.
- `"A function..."` is an optional docstring for the function.

Now that you have the code for the Python bindings, take a look at how you can build this into a Python module.

Build the Python Bindings

The tool you use to build the Python bindings in `PyBind11` is the C++ compiler itself. You may need to modify the defaults for your compiler and operating system.

To begin, you must build the C++ library for which you're creating bindings. For an example this small, you could build the `cppmult` library directly into the Python bindings library. However, for most real-world examples, you'll have a pre-existing library you want to wrap, so you'll build the `cppmult` library separately. The build is a standard call to the compiler to build a shared library:

```
# tasks.py
invoke.run(
    "g++ -O3 -Wall -Werror -shared -std=c++11 -fPIC cppmult.cpp "
    "-o libcppmult.so "
)
```

Running this with `invoke build-cppmult` produces `libcppmult.so`:

```
$ invoke build-cppmult
==================================================
= Building C++ Library
* Complete
```

The build for the Python bindings, on the other hand, requires some special details:

```
# tasks.py
invoke.run(
    "g++ -O3 -Wall -Werror -shared -std=c++11 -fPIC "
    "`python3 -m pybind11 --includes` "
    "-I /usr/include/python3.7 -I .  "
    "{0} "
    "-o {1}`python3.7-config --extension-suffix` "
    "-L. -lcppmult -Wl,-rpath,.".format(cpp_name, extension_name)
)
```

Let's walk through this line-by-line. **Line 3** contains fairly standard C++ compiler flags that indicate several details, including that you want all warnings caught and treated as errors, that you want a shared library, and that you're using C++11.

Line 4 is the first step of the magic. It calls the `pybind11` module to have it produce the proper `include` paths for `PyBind11`. You can run this command directly on the console to see what it does:

```
$ python3 -m pybind11 --includes
-I/home/jima/.virtualenvs/realpython/include/python3.7m
-I/home/jima/.virtualenvs/realpython/include/site/python3.7
```

Your output should be similar but show different paths.

In **line 5** of your compilation call, you can see that you're also adding the path to the Python dev `includes`. While it's recommended that you *don't* link against the Python library itself, the source needs some code from `Python.h` to work its magic. Fortunately, the code it uses is fairly stable across Python versions.

Line 5 also uses `-I .` to add the current directory to the list of `include` paths. This allows the `#include <cppmult.hpp>` line in your wrapper code to be resolved.

Line 6 specifies the name of your source file, which is `pybind11_wrapper.cpp`. Then, on **line 7** you see some more build magic happening. This line specifies the name of the output file. Python has some particular ideas on module naming, which include the Python version, the machine architecture, and other details. Python also provides a tool to help with this called `python3.7-config`:

```
$ python3.7-config --extension-suffix
.cpython-37m-x86_64-linux-gnu.so
```

You may need to modify the command if you're using a different version of Python. Your results will likely change if you're using a different version of Python or are on a different operating system.

The final line of your build command, **line 8**, points the linker at the `libcppmult` library you built earlier. The `rpath` section tells the linker to add information to the shared library to help the operating system find `libcppmult` at runtime. Finally, you'll notice that this string is formatted with the `cpp_name` and the `extension_name`. You'll be using this function again when you build your Python bindings module with `Cython` in the next section.

Run this command to build your bindings:

```
$ invoke build-pybind11
==================================================
= Building C++ Library
* Complete
==================================================
= Building PyBind11 Module
* Complete
```

That's it! You've built your Python bindings with `PyBind11`. It's time to test it out!

Calling Your Function

Similar to the `CFFI` example above, once you've done the heavy lifting of creating the Python bindings, calling your function looks like normal Python code:

```
# pybind11_test.py
import pybind11_example

if __name__ == "__main__":
    # Sample data for your call
    x, y = 6, 2.3

    answer = pybind11_example.cpp_function(x, y)
    print(f"    In Python: int: {x} float {y:.1f} return val {answer:.1f}")
```

Since you used `pybind11_example` as the name of your module in the `PYBIND11_MODULE` macro, that's the name you import. In the `m.def()` call you told `PyBind11` to export the `cppmult` function as `cpp_function`, so that's what you use to call it from Python.

You can test it with `invoke` as well:

```
$ invoke test-pybind11
================================================
= Testing PyBind11 Module
    In cppmul: int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 13.8
```

That's what `PyBind11` looks like. Next, you'll see when and why `PyBind11` is the right tool for the job.

Strengths and Weaknesses

`PyBind11` is focused on C++ instead of C, which makes it different from `ctypes` and `CFFI`. It has several features that make it quite attractive for C++ libraries:

- It supports **classes**.
- It handles **polymorphic subclassing**.
- It allows you to add **dynamic attributes** to objects from Python and many other tools, which would be quite difficult to do from the C-based tools you've examined.

That being said, there's a fair bit of setup and configuration you need to do to get `PyBind11` up and running. Getting the installation and build correct can be a bit finicky, but once that's done, it seems fairly solid. Also, `PyBind11` requires that you use at least C++11 or newer. This is unlikely to be a big restriction for most projects, but it may be a consideration for you.

Finally, the extra code you need to write to create the Python bindings is in C++ and not Python. This may or may not be an issue for you, but it *is* different than the other tools you've looked at here. In the next section, you'll move on to `Cython`, which takes quite a different approach to this problem.

`Cython`

The approach `Cython` takes to creating Python bindings uses a **Python-like language** to define the bindings and then generates C or C++ code that can be compiled into the module. There are several methods for building Python bindings with `Cython`. The most common one is to use `setup` from `distutils`. For this example, you'll stick with the `invoke` tool, which will allow you to play with the exact commands that are run.

How It's Installed

`Cython` is a Python module that can be installed into your virtual environment from PyPI:

```
$ python3 -m pip install cython
```

Again, if you've installed the `requirements.txt` file into your virtual environment, then this will already be there. You can grab a copy of `requirements.txt` by clicking on the link below:

Get Sample Code: Click here to get the sample code you'll use to learn about Python Bindings in this tutorial.

That should have you ready to work with `Cython`!

Calling the Function

To build your Python bindings with `Cython`, you'll follow similar steps to those you used for `CFFI` and `PyBind11`. You'll write the bindings, build them, and then run Python code to call them. `Cython` can support both C and C++. For this example, you'll use the `cppmult` library that you used for the `PyBind11` example above.

Write the Bindings

The most common form of declaring a module in `Cython` is to use a `.pyx` file:

```
# cython_example.pyx
""" Example cython interface definition """

cdef extern from "cppmult.hpp":
    float cppmult(int int_param, float float_param)

def pymult( int_param, float_param ):
    return cppmult( int_param, float_param )
```

There are two sections here:

1. **Lines 3 and 4** tell `Cython` that you're using `cppmult()` from `cppmult.hpp`.
2. **Lines 6 and 7** create a wrapper function, `pymult()`, to call `cppmult()`.

The language used here is a special mix of C, C++, and Python. It will look fairly familiar to Python developers, though, as the goal is to make the process easier.

The first section with `cdef extern...` tells `Cython` that the function declarations below are also found in the `cppmult.hpp` file. This is useful for ensuring that your Python bindings are built against the same declarations as your C++ code. The second section looks like a regular Python function—because it is! This section creates a Python function that has access to the C++ function `cppmult`.

Now that you've got the Python bindings defined, it's time to build them!

Build the Python Bindings

The build process for `Cython` has similarities to the one you used for `PyBind11`. You first run `Cython` on the `.pyx` file to generate a `.cpp` file. Once you've done this, you compile it with the same function you used for `PyBind11`:

```
# tasks.py
def compile_python_module(cpp_name, extension_name):
    invoke.run(
        "g++ -O3 -Wall -Werror -shared -std=c++11 -fPIC "
        "`python3 -m pybind11 --includes` "
        "-I /usr/include/python3.7 -I .  "
        "{0} "
        "-o {1}`python3.7-config --extension-suffix` "
        "-L. -lcppmult -Wl,-rpath,.".format(cpp_name, extension_name)
    )

def build_cython(c):
    """ Build the cython extension module """
    print_banner("Building Cython Module")
    # Run cython on the pyx file to create a .cpp file
    invoke.run("cython --cplus -3 cython_example.pyx -o cython_wrapper.cpp")

    # Compile and link the cython wrapper library
    compile_python_module("cython_wrapper.cpp", "cython_example")
    print("* Complete")
```

You start by running `cython` on your `.pyx` file. There are a few options you use on this command:

- `--cplus` tells the compiler to generate a C++ file instead of a C file.
- `-3` switches `Cython` to generate Python 3 syntax instead of Python 2.
- `-o cython_wrapper.cpp` specifies the name of the file to generate.

Once the C++ file is generated, you use the C++ compiler to generate the Python bindings, just as you did for `PyBind11`. Note that the call to produce the extra `include` paths using the `pybind11` tool is still in that function. It won't hurt anything here, as your source will not need those.

Running this task in `invoke` produces this output:

```
$ invoke build-cython
==================================================
= Building C++ Library
* Complete
==================================================
= Building Cython Module
* Complete
```

You can see that it builds the `cppmult` library and then builds the `cython` module to wrap it. Now you have the `Cython` Python bindings. (Try saying *that* quickly...) It's time to test it out!

Calling Your Function

The Python code to call your new Python bindings is quite similar to what you used to test the other modules:

```
# cython_test.py
import cython_example

# Sample data for your call
x, y = 6, 2.3

answer = cython_example.pymult(x, y)
print(f"    In Python: int: {x} float {y:.1f} return val {answer:.1f}")
```

Line 2 imports your new Python bindings module, and you call `pymult()` on line 7. Remember that the `.pyx` file provided a Python wrapper around `cppmult()` and renamed it to `pymult`. Using invoke to run your test produces the following:

```
$ invoke test-cython
==================================================
= Testing Cython Module
    In cppmul: int: 6 float 2.3 returning  13.8
    In Python: int: 6 float 2.3 return val 13.8
```

You get the same result as before!

Strengths and Weaknesses

`Cython` is a relatively complex tool that can provide you **a deep level of control** when creating Python bindings for either C or C++. Though you didn't cover it in depth here, it provides a Python-esque method for writing code that manually controls the GIL, which can significantly speed up certain types of problems.

That Python-esque language is not quite Python, however, so there's a slight learning curve when you're coming up to speed in figuring out which parts of C and Python fit where.

Other Solutions

While researching this tutorial, I came across several different tools and options for creating Python bindings. While I limited this overview to some of the more common options, there are several other tools I stumbled across. The list below is not comprehensive. It's merely a sampling of other possibilities if one of the above tools doesn't fit your project.

`PyBindGen`

`PyBindGen` generates Python bindings for C or C++ and is written in Python. It's targeted at producing readable C or C++ code, which should simplify debugging issues. It wasn't clear if this has been updated recently, as the documentation lists Python 3.4 as the latest tested version. There have been yearly releases for the last several years, however.

`Boost.Python`

`Boost.Python` has an interface similar to `PyBind11`, which you saw above. That's not a coincidence, as `PyBind11` was based on this library! `Boost.Python` is written in full C++ and supports most, if not all, versions of C++ on most platforms. In contrast, `PyBind11` restricts itself to modern C++.

`SIP`

`SIP` is a toolset for generating Python bindings that was developed for the PyQt project. It's also used by the wxPython project to generate their bindings, as well. It has a code generation tool and an extra Python module that provides support functions for the generated code.

`Cppyy`

`cppyy` is an interesting tool that has a slightly different design goal than what you've seen so far. In the words of the package author:

> "The original idea behind cppyy (going back to 2001), was to allow Python programmers that live in a C++ world access to those C++ packages, without having to touch C++ directly (or wait for the C++ developers to come around and provide bindings)." (Source)

`Shiboken`

`Shiboken` is a tool for generating Python bindings that's developed for the PySide project associated with the Qt project. While it was designed as a tool for that project, the documentation indicates that it's neither Qt- nor PySide-specific and is usable for other projects.

`SWIG`

`SWIG` is a different tool than any of the others listed here. It's a general tool used to create bindings to C and C++ programs for many other languages, not just Python. This ability to generate bindings for different languages can be quite useful in some projects. It, of course, comes with a cost as far as complexity is concerned.

Conclusion

Congrats! You've now had an overview of several different options for creating **Python bindings**. You've learned about marshalling data and issues you need to consider when creating bindings. You've seen what it takes to be able to call a C or C++ function from Python using the following tools:

- `ctypes`
- `CFFI`
- `PyBind11`
- `Cython`

You now know that, while `ctypes` allow you to load a DLL or shared library directly, the other three tools take an extra step, but still create a full Python module. As a bonus, you've also played a little with the `invoke` tool to run command-line tasks from Python. You can get all of the code you saw in this tutorial by clicking the link below:

Get Sample Code: Click here to get the sample code you'll use to learn about Python Bindings in this tutorial.

Now pick your favorite tool and start building those Python bindings! Special thanks to **Loic Domaigne** for the extra technical review of this tutorial.