

Calling C/C++ extensions with ctypes

- [Objectives](#)
- [Why extend Python with C/C++](#)
 - [Pros](#)
 - [Cons](#)
- [Learn the basics](#)
 - [Steps required to call an external function from Python](#)
 - [Calling mysum from Python](#)
 - [Additional explanation](#)
- [Exercises](#)

Objectives

You will learn:

- how to call C/C++ compiled code from Python using the `ctypes` module
- how to compile a C++ extension using `setuptools`

We'll use the code under `cext` . Start by

```
cd cext
```

Why extend Python with C/C++

1. You want to call a function that is implemented in C, C++ or Fortran. This can give you access to a vast collection of libraries so you won't have to reinvent the wheel
2. You have identified a performance bottleneck - reimplementing some parts of your Python code in C, C++ or Fortran could give you a performance boost
3. It makes your code type safe. In contrast to C, C++ and Fortran, Python is not a typed language - you can pass any object to any Python function. This can cause runtime failures in Python which cannot occur in C, C++ or Fortran, as the error would be caught by the compiler

Pros

- A good way to glue Python with an external library
 - Can be used to incrementally migrate code to C/C++
 - Very flexible
- Simpler and easier to maintain than custom C extensions

Cons

- Has a learning curve, one must understand how Python and C work
- Mistakes often lead to segmentation faults, which can be hard to debug

Learn the basics

Let's go to our example of computing the sum of all the elements of an array. In order not to interfere with the scatter code, let's create a directory `mysum_example` and go to that directory:

```
mkdir mysum_example
cd mysum_example
```

Open your editor and copy-paste the following code

```
/**
 * Compute the sum an array
 * @param n number of elements
 * @param array input array
 * @return sum
 */
extern "C" // required when using C++ compiler
long long mysum(int n, int* array) {
    // return type is 64 bit integer
    long long res = 0;
    for (int i = 0; i < n; ++i) {
        res += array[i];
    }
    return res;
}
```

Save the above in file `mysum.cpp`.

Note: the `extern "C"` line ensures that function `mysum` can be called from "C". Because Python is written in C, your external function must be C callable.

To compile `mysum.cpp` we need to write a `setup.py` file. Open your editor, copy-paste the lines

```
from setuptools import setup, Extension

# Compile *mysum.cpp* into a shared library
setup(
    #...
    ext_modules=[Extension('mysum', ['mysum.cpp'],)],
)
```

and save in file `setup.py`. The fact that `mysum.cpp` has the `.cpp` extension indicates that the source file is written in C++.

Compile the code with the command:

```
python setup.py build
```

This will compile the code and produce a shared library under `build/lib.linux-x86_64-3.6`, something like `mysum.cpython-36m-x86_64-linux-gnu.so`. The extension `.so` indicates that the file is a shared library (also called dynamic-link library or shared object). The advantage of creating a

shared library over a static library is that in the former the Python interpreter needs not be recompiled. The good news is that `setuptools` knows how to compile shared libraries so you won't have to worry about the details.

Notes:

- by convention this file should be named `setup.py`
- a more realistic example might have `include` directories and libraries listed in `setup.py` if the C++ extension depends on external packages. An example of a `setup.py` file can be found [here](#).

Steps required to call an external function from Python

To call `mysum` from Python we'll use the `ctypes` module. The steps are:

1. use function `CDLL` to open the shared library. `CDLL` expects the path to the shared library and returns a shared library object.
2. tell the argument and result types of the function. The argument types are listed in members `argtypes` (a list) and `restype`, respectively. Use for instance `ctypes.c_int` for a C `int`. See table below to find out how to translate other C types to their corresponding `ctypes` objects.
3. call the function, casting the Python objects into ctypes objects **if required**. The table below shows how you can cast some common C/C++ types in corresponding Python objects, which can be handed over to an external C/C++ function

Translation table for some Python and C/C++ types

The following table can be used to translate some common types between Python and C:

Python	C/C++ type	Comments
<code>None</code>	<code>NULL</code>	
<code>ctypes.char_p</code>	<code>char*</code>	
<code>ctypes.c_int</code>	<code>int</code>	No need to cast
<code>ctypes.c_longlong</code>	<code>long</code> <code>long</code>	
<code>ctypes.c_double</code>	<code>double</code>	
<code>numpy.ctypeslib.ndpointer(dtype=numpy.float64)</code>	<code>double*</code>	pass a numpy array of type <code>numpy.float64</code>
<code>numpy.ctypeslib.ndpointer(dtype=numpy.int32)</code>	<code>int*</code>	pass a numpy array of type <code>numpy.int32</code>

For a complete list of C to ctypes type mapping see the Python [documentation](#).

Calling mysum from Python

Let's return to our `mysum` C++ function, which we would like to call from Python:

```
import ctypes
import numpy
import glob

# find the shared library, the path depends on the platform and Python version
libfile = glob.glob('build/*/mysum*.so')[0]

# 1. open the shared library
mylib = ctypes.CDLL(libfile)

# 2. tell Python the argument and result types of function mysum
```

```

mylib.mysum.restype = ctypes.c_longlong
mylib.mysum.argtypes = [ctypes.c_int,
                        numpy.ctypeslib.ndpointer(dtype=numpy.int32)]

array = numpy.arange(0, 100000000, 1, numpy.int32)

# 3. call function mysum
array_sum = mylib.mysum(len(array), array)

print('sum of array: {}'.format(array_sum))

```

Additional explanation

- By default, arguments are passed by value. To pass an array of ints (`int*`), specify `numpy.ctypeslib.ndpointer(dtype=numpy.int32)` in the `argtypes` list. You can declare `double*` similarly by using `numpy.ctypeslib.ndpointer(dtype=numpy.float64)`
- Strings will need to be converted to byte strings in Python 3 (`str(mystring).encode('ascii')`)
- Passing by reference, for instance `int&` can be achieved using `ctypes.byref(myvar_t)` with `myvar_t` of type `ctypes.c_int`
- Numpy arrays of type `numpy.int` have precision `numpy.int64` so make sure to create an array of type `numpy.int32`, which has the same precision as C `int`.
- When passing arrays, it is possible to specify extra restrictions on the numpy arrays at the interface level, for example the number of dimensions the array should have or its shape. If an array passed in as an argument does not meet the specified requirements and exception will be raised. A full list of possible options can be found in the `numpy.ctypeslib.ndpointer` [documentation](#).

Exercises


We've created a version of `scatter.py` that compute the scattered wave from a contour segment in C++. Compile the code using `python setup.py build`. (Make sure you have the `BOOST_DIR` environment variable set as described [here](#)). The code runs faster than the pure Python code under `original/`; however, there is still room for improvement. Your task will be to replace the computation in Python function `isInsideContour` defined in `scatter.py` with the C++ version implemented in `src/is_inside_contour.cpp`.

- run and time/profile `scatter.py`, making note of the checksum
- look into `src/is_inside_contour.h` to determine the interface of the function
- in `setup.py`, add `src/is_inside_contour.cpp` to the `wave` library extension
- define the C++ function interface in `scatter.py`
- modify `scatter.py` to call the C++ function
- re-run and time/profile `scatter.py`, making sure the checksum has not changed

« Numba

Multiprocessing »

New Zealand eScience Infrastructure - Performance Optimisation Training

New Zealand eScience Infrastructure  [nesi_nz](#)
Performance Optimisation Training
[ing@nesi.org.nz](mailto:training@nesi.org.nz)

Training materials for a hands-on workshop on
Performance Optimisation



Support



The Power Behind Researchers