Calling C/C++ from Python? [closed]

Asked 12 years, 11 months ago Active 2 months ago Viewed 480k times

561

374

Closed. This question needs to be more focused. It is not currently accepting answers.

Want to improve this question? Update the question so it focuses on one problem only by editing this post.

Closed 11 months ago.

Improve this question

What would be the quickest way to construct a Python binding to a C or C++ library?

(I am using Windows if this matters.)

c++ python c

Share Edit Follow Flag

edited Oct 4 '17 at 22:55

Peter Mortensen
28.8k 21 95 123

asked Sep 28 '08 at 5:34

shoosh
71.6k 50 201 313

12 Answers

Active Oldest Votes

714

ctypes module is part of the standard library, and therefore is more stable and widely available than swig, which always tended to give me problems.

With ctypes, you need to satisfy any compile time dependency on python, and your binding will work on any python that has ctypes, not just the one it was compiled against.

Suppose you have a simple C++ example class you want to talk to in a file called foo.cpp:

```
#include <iostream>

class Foo{
    public:
        void bar(){
            std::cout << "Hello" << std::endl;
        }
};
```

Since ctypes can only talk to C functions, you need to provide those declaring them as extern "C"

```
extern "C" {
    Foo* Foo_new(){ return new Foo(); }
    void Foo_bar(Foo* foo){ foo->bar(); }
}
```

Next you have to compile this to a shared library

```
g++ -c -fPIC foo.cpp -o foo.o
g++ -shared -Wl,-soname,libfoo.so -o libfoo.so  foo.o
```

And finally you have to write your python wrapper (e.g. in fooWrapper.py)

```
from ctypes import cdll
lib = cdll.LoadLibrary('./libfoo.so')

class Foo(object):
```

```
f = Foo()
f.bar() #and you will see "Hello" on the screen
```

Share Edit Follow Flag

edited Jun 7 '20 at 15:48
mirekphd
1,886 14 23

answered Sep 28 '08 at 10:53

Florian Bösch
26.2k 11 45 52

14 This is pretty much what boost.python does for you in a single function call. – Martin Beckett Sep 29 '08 at 16:36

220 ctypes is in the python standard library, swig and boost are not. Swig and boost rely on extension modules and are therefore tied to python minor versions which indepentent shared objects are not. building a swig or boost wrappers can be a pain, ctypes makes no build requirements. – Florian Bösch Sep 29 '08 at 22:42

29 boost relies on voodoo template magic and an entirely custom build system, ctypes relies on simplicity. ctypes is dynamic, boost is static. ctypes can handle different versions of libraries. boost cannot. – Florian Bösch Sep 29 '08 at 22:44

37 On Windows I've had to specify __declspec(dllexport) in my function signatures for Python to be able to see them. From the above example this would correspond to: `extern "C" { __declspec(dllexport) Foo* Foo_new(){ return new Foo(); } __declspec(dllexport) void Foo_bar(Foo* foo){ foo->bar(); } }` – Alan Macdonald Nov 30 '11 at 11:12

17 Don't forget to delete the pointer afterwards by e.g. providing a `Foo_delete` function and calling it either from a python destructor or wrapping the object in a resource. – Adversus Nov 4 '15 at 8:34

187

You should have a look at Boost.Python. Here is the short introduction taken from their website:

> The Boost Python Library is a framework for interfacing Python and C++. It allows you to quickly and seamlessly expose C++ classes functions and objects to Python, and vice-versa, using no special tools -- just your C++ compiler. It is designed to wrap C++ interfaces non-intrusively, so that you should not have to change the C++ code at all in order to wrap it, making Boost.Python ideal for exposing 3rd-party libraries to Python. The library's use of advanced metaprogramming techniques simplifies its syntax for users, so that wrapping code takes on the look of a kind of declarative interface definition language (IDL).

Share Edit Follow Flag

edited Oct 4 '17 at 22:58

Peter Mortensen
28.8k 21 95 123

answered Sep 28 '08 at 7:51

Ralph
4,984 1 19 18

1 Boost.Python is one of the more user-friendly libraries in Boost, for a simple function call API it is quite straightforward and provides boilerplate you'd have to write yourself. It's a bit more complicated if you want to expose an object-oriented API. – jwfearn Sep 28 '08 at 16:39

21 Boost.Python is the worst thing imaginable. For every new machine and with every upgrade it goes with linking problems. – miller Sep 18 '18 at 9:58

27 Nearly 11 years later time for some contemplation about the quality of this answer? – J Evans May 23 '19 at 19:59

4 Is this still the best approach to interface python and c++ ? – tushaR Jun 17 '19 at 2:28

12 Maybe you can try pybind11 which is lightweight compared to boost. – jdhao Jul 16 '19 at 12:46

58

The quickest way to do this is using SWIG.

Example from SWIG tutorial:

```
/* File : example.c */
int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}
```

Interface file:

```
/* example.i */
%module example
%{
/* Put header files here or function declarations like below */
extern int fact(int n);
%}

extern int fact(int n);
```

Building a Python module on Unix:

```
swig -python example.i
gcc -fPIC -c example.c example_wrap.c -I/usr/local/include/python2.7
gcc -shared example.o example_wrap.o -o _example.so
```

Usage:

```
>>> import example
>>> example.fact(5)
120
```

Note that you have to have python-dev. Also in some systems python header files will be in /usr/include/python2.7 based on the way you have installed it.

proxy classes in the target language — exposing the underlying functionality in a very natural manner.

Share Edit Follow Flag

edited Dec 14 '15 at 17:39

mmohaveri
418 7 19

answered Sep 28 '08 at 5:44

Ben Hoffstein
98.8k 8 101 119

53

I started my journey in the Python <-> C++ binding from this page, with the objective of linking high level data types (multidimensional STL vectors with Python lists) :-)

Having tried the solutions based on both ctypes and boost.python (and not being a software engineer) I have found them complex when high level datatypes binding is required, while I have found SWIG much more simple for such cases.

This example uses therefore SWIG, and it has been tested in Linux (but SWIG is available and is widely used in Windows too).

The objective is to make a C++ function available to Python that takes a matrix in form of a 2D STL vector and returns an average of each row (as a 1D STL vector).

The code in C++ ("code.cpp") is as follow:

```
#include <vector>
#include "code.h"

using namespace std;

vector<double> average (vector< vector<double> > i_matrix) {

  // Compute average of each row..
  vector <double> averages;
  for (int r = 0; r < i_matrix.size(); r++){
    double rsum = 0.0;
    double ncols= i_matrix[r].size();
    for (int c = 0; c< i_matrix[r].size(); c++){
      rsum += i_matrix[r][c];
    }
    averages.push_back(rsum/ncols);
  }
  return averages;
}
```

The equivalent header ("code.h") is:

```
#ifndef _code
#define _code
```

We first compile the C++ code to create an object file:

```
g++ -c -fPIC code.cpp
```

We then define a SWIG interface definition file ("code.i") for our C++ functions.

```
%module code
%{
#include "code.h"
%}
%include "std_vector.i"
namespace std {

  /* On a side note, the names VecDouble and VecVecdouble can be changed, but
the order of first the inner vector matters! */
  %template(VecDouble) vector<double>;
  %template(VecVecdouble) vector< vector<double> >;
}

%include "code.h"
```

Using SWIG, we generate a C++ interface source code from the SWIG interface definition file..

```
swig -c++ -python code.i
```

We finally compile the generated C++ interface source file and link everything together to generate a shared library that is directly importable by Python (the "_" matters):

```
g++ -c -fPIC code_wrap.cxx  -I/usr/include/python2.7 -I/usr/lib/python2.7
g++ -shared -Wl,-soname,_code.so -o _code.so code.o code_wrap.o
```

Share Edit Follow Flag

edited Oct 4 '17 at 23:05

Peter Mortensen
28.8k 21 95 123

answered May 26 '14 at 8:30

Antonello
4,153 3 22 49

A real case implementation where in the C++ code stl vectors are passed as non const references and hence available by python as output parameters: lobianco.org/antonello/personal:portfolio:portopt – Antonello Jun 17 '14 at 7:46

51

There is also `pybind11` , which is like a lightweight version of Boost.Python and compatible with all modern C++ compilers:

https://pybind11.readthedocs.io/en/latest/

Share Edit Follow Flag

edited Jan 18 '18 at 15:43

answered Jul 23 '16 at 13:53

Tom Wenseleers
6,772 5 48 99

9 **Today**!! *2020* This should be the top answer! It is a template header only library. Many huge relevant projects recommends it, like `Pytorch` pytorch.org/tutorials/advanced/cpp_extension.html Also fully works on `VS Community` Windows – eusoubrasileiro Feb 5 '20 at 12:41

2 Please also include some examples, to make it more useful than a link-only answer. – Franklin Yu Jul 16 '20 at 21:25

1 This actually looks amazing! it's a shame I just found this. I am thinking it is worth rewriting code to accomodate this. – LUser Aug 1 '20 at 10:45

2 I've shown a minimal runnable example at: stackoverflow.com/a/60374990/895245 to showcase how awesome it is. – Ciro Santilli 新疆再教育营六四事件法轮功郝海东 Aug 26 '20 at 11:26

29

For modern C++, use cppyy: http://cppyy.readthedocs.io/en/latest/

It's based on Cling, the C++ interpreter for Clang/LLVM. Bindings are at run-time and no additional intermediate language is necessary. Thanks to Clang, it supports C++17.

Install it using pip:

```
$ pip install cppyy
```

For small projects, simply load the relevant library and the headers that you are interested in. E.g. take the code from the ctypes example is this thread, but split in header and code sections:

```
$ cat foo.h
class Foo {
public:
    void bar();
};

$ cat foo.cpp
#include "foo.h"
#include <iostream>

void Foo::bar() { std::cout << "Hello" << std::endl; }
```

Compile it:

```
$ g++ -c -fPIC foo.cpp -o foo.o
$ g++ -shared -Wl,-soname,libfoo.so -o libfoo.so  foo.o
```

and use it:

```
$ python
>>> import cppyy
>>> cppyy.include("foo.h")
>>> cppyy.load_library("foo")
```

users of installed packages can simply run:

```
$ python
>>> import cppyy
>>> f = cppyy.gbl.Foo()
>>> f.bar()
Hello
>>>
```

Thanks to LLVM, advanced features are possible, such as automatic template instantiation. To continue the example:

```
>>> v = cppyy.gbl.std.vector[cppyy.gbl.Foo]()
>>> v.push_back(f)
>>> len(v)
1
>>> v[0].bar()
Hello
>>>
```

Note: I'm the author of cppyy.

Share Edit Follow Flag

edited Mar 6 '18 at 18:21

answered Mar 5 '18 at 23:02

Wim Lavrijsen
2,458 1 7 18

3 It doesn't, really: Cython is a Python-like programming language to write C extension modules for Python (the Cython code gets translated into C, together with the necessary C-API boilerplate). It provides some basic C++ support. Programming with cppyy only involves Python and C++, no language extensions. It's fully run-time and does not generate offline code (lazy generation scales a lot better). It targets modern C++ (incl. automatic template instantiations, moves, initializer_lists, lambda's, etc., etc.) and PyPy is supported natively (i.e. not through the slow C-API emulation layer). – Wim Lavrijsen Apr 21 '18 at 21:57

14

I think cffi for python can be an option.

> The goal is to call C code from Python. You should be able to do so without learning a 3rd language: every alternative requires you to learn their own language (Cython, SWIG) or API (ctypes). So we tried to assume that you know Python and C and minimize the extra bits of API that you need to learn.

http://cffi.readthedocs.org/en/release-0.7/

Share Edit Follow Flag

answered Nov 5 '13 at 10:39

mrgloom

15.9k 26 130 239

2 I think this can only call c (not c++), still +1 (I really like cffi). – Andy Hayden Nov 4 '14 at 3:01

pybind11 minimal runnable example

10

pybind11 was previously mentioned at https://stackoverflow.com/a/38542539/895245 but I would like to give here a concrete usage example and some further discussion about implementation.

All and all, I highly recommend pybind11 because it is really easy to use: you just include a header and then pybind11 uses template magic to inspect the C++ class you want to expose to Python and does that transparently.

The downside of this template magic is that it slows down compilation immediately adding a few seconds to any file that uses pybind11, see for example the investigation done on this issue. PyTorch agrees. A proposal to remediate this problem has been made at: https://github.com/pybind/pybind11/pull/2445

Here is a minimal runnable example to give you a feel of how awesome pybind11 is:

class_test.cpp

```
#include <string>

#include <pybind11/pybind11.h>

struct ClassTest {
    ClassTest(const std::string &name, int i) : name(name), i(i) { }
    void setName(const std::string &name_) { name = name_; }
    const std::string getName() const { return name + "z"; }
    void setI(const int i) { this->i = i; }
    const int getI() const { return i + 1; }
    std::string name;
    int i;
};

namespace py = pybind11;

PYBIND11_PLUGIN(class_test) {
    py::module m("my_module", "pybind11 example plugin");
    py::class_<ClassTest>(m, "ClassTest")
        .def(py::init<const std::string &, int>())
        .def("setName", &ClassTest::setName)
        .def("getName", &ClassTest::getName)
        .def_readwrite("name", &ClassTest::name)
        .def("setI", &ClassTest::setI)
```

```
#!/usr/bin/env python3

import class_test

my_class_test = class_test.ClassTest("abc", 1);
print(my_class_test.getName())
print(my_class_test.getI())
my_class_test.setName("012")
my_class_test.setI(2)
print(my_class_test.getName())
print(my_class_test.getI())
assert(my_class_test.getName() == "012z")
assert(my_class_test.getI() == 3)
```

Compile and run:

```
#!/usr/bin/env bash
set -eux
sudo apt install pybind11-dev
g++ `python3-config --cflags` -shared -std=c++11 -fPIC class_test.cpp \
  -o class_test`python3-config --extension-suffix` `python3-config --libs`
./class_test_main.py
```

Stdout output:

```
abcz
2
012z
3
```

If we tried to use a wrong type as in:

```
Invoked with: <my_module.ClassTest object at 0x7f2980254fb0>, 'abc'
```

This example shows how pybind11 allows you to effortlessly expose the `ClassTest` C++ class to Python!

Notably, Pybind11 automatically understands from the C++ code that `name` is an `std::string`, and therefore should be mapped to a Python `str` object.

Compilation produces a file named `class_test.cpython-36m-x86_64-linux-gnu.so` which `class_test_main.py` automatically picks up as the definition point for the `class_test` natively defined module.

Perhaps the realization of how awesome this is only sinks in if you try to do the same thing by hand with the native Python API, see for example this example of doing that, which has about 10x more code: https://github.com/cirosantilli/python-cheat/blob/4f676f62e87810582ad53b2fb426b74eae52aad5/py_from_c/pure.c On that example you can see how the C code has to painfully and explicitly define the Python class bit by bit with all the information it contains (members, methods, further metadata...). See also:

- Can python-C++ extension get a C++ object and call its member function?
- Exposing a C++ class instance to a python embedded interpreter
- A full and minimal example for a class (not method) with Python C Extension?
- Embedding Python in C++ and calling methods from the C++ code with Boost.Python
- Inheritance in Python C++ extension

> 3.x, or PyPy2.7 >= 5.7) and the C++ standard library. This compact implementation was possible thanks to some of the new C++11 language features (specifically: tuples, lambda functions and variadic templates). Since its creation, this library has grown beyond Boost.Python in many ways, leading to dramatically simpler binding code in many common situations.

pybind11 is also the only non-native alternative hightlighted by the current Microsoft Python C binding documentation at: https://docs.microsoft.com/en-us/visualstudio/python/working-with-c-cpp-python-in-visual-studio?view=vs-2019 (archive).

Tested on Ubuntu 18.04, pybind11 2.0.1, Python 3.6.8, GCC 7.4.0.

Share Edit Follow Flag

edited Jun 17 at 21:41

answered Feb 24 '20 at 11:28

Ciro Santilli 新疆再教育营六四事件法轮功郝海东

270k 76 1005 804

8

The question is how to call a C function from Python, if I understood correctly. Then the best bet are Ctypes (BTW portable across all variants of Python).

```
>>> from ctypes import *
>>> libc = cdll.msvcrt
>>> print libc.time(None)
1438069008
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
>>> printf("%d bottles of beer\n", 42)
42 bottles of beer
19
```

For a detailed guide you may want to refer to my blog article.

Share Edit Follow Flag

edited Aug 28 '15 at 8:08
Palec
10.7k 7 53 118

answered Aug 28 '15 at 6:38
Jadav Bheda
4,371 1 24 26

It may be worth noting that while ctypes are portable, your code requires a Windows-specific C library. – Palec Aug 28 '15 at 8:32

6

Cython is definitely the way to go, unless you anticipate writing Java wrappers, in which case SWIG may be preferable.

I recommend using the `runcython` command line utility, it makes the process of using Cython extremely easy. If you need to pass structured data to C++, take a look at Google's protobuf library, it's very convenient.

Here is a minimal examples I made that uses both tools:

https://github.com/nicodjimenez/python2cpp

Hope it can be a useful starting point.

Share Edit Follow Flag

answered Dec 29 '15 at 17:27

nicodjimenez
997 15 12

6

I love cppyy, it makes it very easy to extend Python with C++ code, dramatically increasing performance when needed.

It is powerful and frankly very simple to use,

here it is an example of how you can create a numpy array and pass it to a class member function in C++.

cppyy_test.py

```
import cppyy
import numpy as np
cppyy.include('Buffer.h')


s = cppyy.gbl.Buffer()
numpy_array = np.empty(32000, np.float64)
s.get_numpy_array(numpy_array.data, numpy_array.size)
print(numpy_array[:20])
```

Buffer.h

```
struct Buffer {
  void get_numpy_array(double *ad, int size) {
    for( long i=0; i < size; i++)
        ad[i]=i;
  }
};
```

You can also create a Python module very easily (with CMake), this way you will avoid recompile the C++ code all the times.

Share Edit Follow Flag

edited Jun 9 '20 at 21:58

philn

376 3 13

answered Dec 19 '19 at 9:56

Garfield

61 1 1

5

First you should decide what is your particular purpose. The official Python documentation on extending and embedding the Python interpreter was mentioned above, I can add a good overview of binary extensions. The use cases can be divided into 3 categories:

- **accelerator modules**: to run faster than the equivalent pure Python code runs in CPython.
- **wrapper modules**: to expose existing C interfaces to Python code.
- **low level system access**: to access lower level features of the CPython runtime, the operating system, or the underlying hardware.

In order to give some broader perspective for other interested and since your initial question is a bit vague ("to a C or C++ library") I think this information might be interesting to you. On the link above you can read on disadvantages of using binary extensions and its alternatives.

Apart from the other answers suggested, if you want an accelerator module, you can try Numba. It works "by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool)".

Share Edit Follow Flag

edited May 15 '19 at 11:17

answered Apr 24 '15 at 17:23

Yaroslav Nikitenko

1,402 1 19 25