IMPERIAL

Calling C++ code from Python

Advanced Programming

ESE MSc 2024

Thomas M. Davison

What will we see in today's session?

Learning Objectives:

- Understand why you might want to call C++ code from python
- See that there are several methods for doing this
- Follow a detailed example of how to achieve this using the **ctypes** python library in both MacOS (VS Code/gcc) and Windows (Visual Studio/MSVC)

Note: After today's class, you might consider that this is a useful tool for your image filters group project. You are welcome to use python to write a user interface for the project, but **it is not required to achieve a good grade**

Several reasons we might want an interface between python and C++

- **Speed advantage:** C++ code is often (usually!) quicker to execute than equivalent python code. Writing critical parts of your code in C++ can help improve performance.

- **User friendly:** Many people find that python is much more user friendly (both for using and developing)
  - Quick to modify functions/UI, etc. without needing to recompile
  - Can use python interactively, in notebooks, etc.

- **Existing libraries:** If you are working with an existing library from one language, it may be better/easier/quicker to write an interface than reinventing the wheel

Combining these elements can be a powerful way to get the most from your library/program/software package

As usual, multiple ways to achieve this

Boost.Python

- Requires external Boost C++ libraries, although widely used
- High level C++ API for interfacing with Python

Cython

- Requires separate Cython compiler
- Need to learn a little new syntax
- Mix Python and C++ seamlessly

Python-C API

[see the C-primer for an example!]

- Low-level access
- Fine grained control
- Complex code

ctypes

- Python standard library
- Easy to use (for simple interfaces)
- Limited to calling functions from shared libraries
- Low-level control

This sounds hard… what's involved?

- Not too much effort, once you know the syntax

- In general, you can keep most of the C++ code the same, and write a separate interface wrapper in C
  - When compiled into a shared library, this is what python will be able to see

- In python, just need to write a class which is aware of the C++ class methods

- Also need to manually define data types in python for the arguments and return values for all functions (methods) in the C++ class.

Writing your external C interface

- Python can’t interface directly with C++ code, but it can with C code.
- So, we need to write an interface that it can understand
- Every method of your class (that you want python to be able to see/interact with) should be declared in this way, including constructors and destructors
- Put it all within an `extern “C” {}` block.

For a simple class with a constructor, destructor, getter and setter:

```
class MyClass {
public:
    MyClass();
    ~MyClass();
    void setInt(int value);
    int getInt();
}

extern "C" {
    MyClass* MyClass_new() { return new MyClass(); }
    void MyClass_delete(MyClass* myClass) { delete myClass; }
    void MyClass_setInt(MyClass* myClass, const int value) {
        myClass->setInt(value);
    }
    const int MyClass_getInt(MyClass* myClass) { return myClass->getInt(); }
}
```

With our interface written, we need to compile a shared library

With the gcc compilers (e.g. on Macs):

Assuming your class is written in **myclass.cpp**, and your wrapper is in **myclassWrapper.cpp**

```
g++ -shared -o libmyclass.so -fPIC myclass.cpp myclassWrapper.cpp
```

In this case, **libmyclass.so** is the file that will be imported in python

With the MSVC compiler (Visual Studio on Windows):

- Need to set “Configuration Type” to DLL
- Change the output directory and target name as appropriate
- Need to add an extra keyword (**declspec**) before all declarations in the **extern “C”** block:

```
__declspec(dllexport) MyClass* MyClass_new() { return new MyClass(); }
__declspec(dllexport) void MyClass_delete(MyClass* myClass) { delete myClass; }
__declspec(dllexport) void MyClass_setInt(MyClass* myClass, const int value) {
      myClass->setInt(value);
}
```

Writing your python interface

First, import the **ctypes** module (or the relevant parts of it):

```
from ctypes import (cdll, byref,
                    c_int, c_float,
                    c_char_p, c_void_p,
                    POINTER, c_size_t)
```

Import your shared library using:

```
ctypes.cdll.LoadLibrary
```

e.g.:

```
lib = cdll.LoadLibrary('./libmyclass.so’)
```

(alternatively, slightly shorter, but equivalent):

```
from ctypes import CDLL
lib = CDLL('./libmyclass.so')
```

ctypes type	C type	Python type
c_bool	_Bool	bool
c_char	char	1-character bytes object
c_byte	char	int
c_ubyte	unsigned char	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_size_t	size_t	int
c_float	float	float
c_double	double	float
c_char_p	char* (NUL terminated)	bytes object or None
c_void_p	void*	int or None

Full list here:
https://docs.python.org/3/library/ctypes.html#fundamental-data-types

Define the argument and return types for each of the methods in your class using the types imported from ctypes

Method names from **`extern "C"`** block

Should define **`argtypes`** and **`restype`** for all methods

```
lib.MyClass_new.argtypes = []
lib.MyClass_new.restype = c_void_p

lib.MyClass_delete.argtypes = [c_void_p]
lib.MyClass_delete.restype = None

lib.MyClass_setInt.argtypes = [c_void_p, c_int]
lib.MyClass_setInt.restype = None

lib.MyClass_getInt.argtypes = [c_void_p]
lib.MyClass_getInt.restype = c_int
```

```
extern "C" {
    MyClass* MyClass_new()
    {
        return new MyClass();
    }
    void MyClass_delete(MyClass* myClass)
    {
        delete myClass;
    }
    void MyClass_setInt(MyClass* myClass, const int value)
    {
        myClass->setInt(value);
    }
    const int MyClass_getInt(MyClass* myClass)
    {
        return myClass->getInt();
    }
}
```

Create a python class with methods corresponding to your C++ class methods

```
class MyClass(object):

    def __init__(self):
        self.obj = lib.MyClass_new()

    def __del__(self):
        lib.MyClass_delete(self.obj)

    def setInt(self, a):
         lib.MyClass_setInt(self.obj, a)

     def getInt(self):
        return lib.MyClass_getInt(self.obj)
```

```
lib.MyClass_new.argtypes = []
lib.MyClass_new.restype = c_void_p

lib.MyClass_delete.argtypes = [c_void_p]
lib.MyClass_delete.restype = None

lib.MyClass_setInt.argtypes = [c_void_p, c_int]
lib.MyClass_setInt.restype = None

lib.MyClass_getInt.argtypes = [c_void_p]
lib.MyClass_getInt.restype = c_int
```

You can the interface with the class like any other python class

For our example:

```
cls = MyClass()

cls.setInt(5)

myInt = cls.getInt()  # sets myInt to 5

print(myInt)

del cls
```

IMPERIAL

Let’s go to live coding to see how we might build a class and interface using what we’ve learned