Res. Lett. Inf. Math. Sci., 2009, Vol. **13**, pp. 1–7
Available online at http://iims.massey.ac.nz/research/letters/

Automatic C Library Wrapping – Ctypes from the Trenches

Guy K. Kloss
Computer Science
Institute of Information & Mathematical Sciences
Massey University at Albany, Auckland, New Zealand
Email: G.Kloss@massey.ac.nz

At some point of time many Python developers – at least in computational science – will face the situation that they want to interface some natively compiled library from Python. For binding native code to Python by now a larger variety of tools and technologies are available. This paper focuses on wrapping shared C libraries, using Python's default *Ctypes*. Particularly tools to ease the process (by using code generation) and some best practises will be stressed. The paper will try to tell a "step–by–step" story of the wrapping and development process, that should be transferable to similar problems.

Keywords: Python, Ctypes, wrapping, automation, code generation.

1 Introduction

One of the grand fundamentals in software engineering is to use the tools that are best suited for a job, and not to prematurely decide on an implementation. That is often easier said than done, in the light of some complimentary requirements (e. g. rapid/easy implementation vs. needed speed of execution or vs. low level access to hardware). The "traditional" way [1] of binding native code to Python through *extending* or *embedding* is quite tedious and requires lots of manual coding in C. This paper presents an approach using the *Ctypes* package [2], which is by default part of Python since version 2.5.

As an example the creation of a wrapper for the Little CMS colour management library [3] is outlined. The library offers excellent features, and ships with "official" Python bindings (using *SWIG* [4]), but unfortunately with several shortcomings (incompleteness, un-Pythonic API, complex to use, etc.). So out of need and frustration the initial steps towards alternative Python bindings were undertaken.

An alternative would be to fix or improve the bindings using *SWIG,* or to use one of a variety of binding tools. The field has been limited to tools that are widely in use today within the community, and that are promising to be future proof as well as not overly complicated to use. These are the contestants with (very brief) notes for use cases that suit their particular strengths:

- Use *Ctypes* [2], if you want to wrap pure C code very easily.
- Use *Boost.Python* [5,6], if you want to create a more complete API for C++ that also reflects the object oriented nature of your native code, including inheritance into Python, etc.
- Use *cython* [7], if you want to easily speed up and migrate code from Python to speedier native code (Mixing is possible!).
- Use *SWIG* [4], if you want to wrap your code against several dynamic languages.

Of course, wrapper code can be written manually, in this case directly using *Ctypes*. This paper does not provide a tutorial on how *Ctypes* is used. The reader should be familiar with this package when attempting to undertake serious library wrapping. The *Ctypes tutorial* and *Ctypes reference* on the project web site [2] are an excellent starting point for this. For extensive libraries and robustness towards an evolving API, code generation proved to be a good approach over manual editing. Code generators exist for *Boost.Python* as well as for *Ctypes* to ease the process of wrapping: *Py++* [8] (for *Boost.Python*) and *CtypesLib's* [2] `h2xml.py` and `xml2py.py`.

Three main reasons have influenced the decision to approach this project using *ctypes:*

- Ubiquity of the binding approach, as *Ctypes* is part of the default distribution.
- No compilation of native code to libraries is necessary. Additionally, this relieves one from installing a number of development tools, and the library wrapper can be approached in a platform independent way.
- The availability of a code generator to automate large portions of the wrapper implementation process for ease and robustness against changes.

The next section of this paper will first introduce a simple C example. This example is later migrated to Python code through the various "incarnations" of the Python wrapper throughout the paper. Sect. 3 introduces how to facilitate the C library code from Python, in this case through code generation. Sect. 4 explains how to refine the generated code to meet the desired functionality of the wrapper. The library is anything but "Pythonic," so Sect. 5 explains an object oriented Façade API for the library that features "qualities we love."

This paper only outlines some interesting fundamentals of the wrapper building process. Please refer to the source code for more precise details [9].

2 The Example

The sample code (listing in Fig. 1) aims to convert image data from device dependent colour information to a standardised colour space. The input profile results from a device specific characterisation of a Hewlett Packard ScanJet (in the ICC profile `HPSJTW.ICM`). The output is in the standard conformant sRGB output colour space as it is used for the majority of displays on computers. For this a built-in profile from *LittleCMS* is used.

Input and output are characterised through so called "ICC profiles." For the input profile the characterisation is read from a file (line 8), and a built in output profile is used (line 9). The transformation object is set up using the profiles (lines 11–13), specifying the colour encoding in the in- and output as well as some further parameters not worth discussing here. In the **`for`** loop (lines 15–21) the image data is transformed line by line, operating on the number of pixels used per line (necessary as array rows are often padded).

The goal is to provide a suitable and easy to use API to perform the same task in Python.

3 Code Generation

Wrapping C data types, functions, constants, etc. with *Ctypes* is not particularly difficult. The tutorial, project web site and documentation on the wiki introduce this concept quite well. But in the presence of an existing larger library, manual wrapping can be tedious and error prone, as well as hard to keep consistent with the library in case of changes. This is especially true when the library is maintained by someone else. Therefore, it is advisable to generate the wrapper code.

Thomas Heller, the author of *Ctypes* has implemented a corresponding project *CtypesLib* that includes tools for code generation. The tool chain consists of two parts, the parser (for header files) and the code generator.

```c
#include "lcms.h"

int correctColour(void) {
    cmsHPROFILE inProfile, outProfile;
    cmsHTRANSFORM myTransform;
    int i;

    inProfile  = cmsOpenProfileFromFile("HPSJTW.ICM", "r");
    outProfile = cmsCreate_sRGBProfile();

    myTransform = cmsCreateTransform(inProfile, TYPE_RGB_8,
                                     outProfile, TYPE_RGB_8,
                                     INTENT_PERCEPTUAL, 0);

    for (i = 0; i < scanLines; i++) {
        /* Skipped pointer handling of buffers. */
        cmsDoTransform(myTransform,
                       pointerToYourInBuffer,
                       pointerToYourOutBuffer,
                       numberOfPixelsPerScanLine);
    }

    cmsDeleteTransform(myTransform);
    cmsCloseProfile(inProfile);
    cmsCloseProfile(outProfile);

    return 0;
}
```

Figure 1: Example in C using the *LittleCMS* library directly.

3.1 Parsing the Header File

The C header files are parsed by the tool `h2xml`. In the background it uses GCCXML, a GCC compiler that parses the code and generates an XML tree representation. Therefore, usually the same compiler that builds the binary of the library can be used to analyse the sources for the code generation. Alternative parsers often have problems determining a 100 % proper interpretation of the code. This is particularly true in the case of C code containing pre-processor macros, which can "commit" massively complex things.

3.2 Generating the Wrapper

In the next stage the parser tree in XML format is taken to generate the binding code in Python using *Ctypes*. This task is performed by the `xml2py` tool. The generator can be configured in its actions by means of switches passed to it. Of particular interest here are the `-k` and the `-r` switches. The former defines the kind of types to include in the output. In this case the `#define`s, functions, structure and union definitions are of interest, yielding `-kdfs`. Note: Dependencies are resolved automatically. The `-r` switch takes a regular expression the generator uses to identify symbols to generate code for. The full argument list is shown in the listing in Fig. 2 (lines 11–15). The generated code is written to a Python module, in this case `_lcms`. It is made private by convention (leading underscore) to indicate that it is *not* to be used or modified directly.

3.3 Automating the Generator

Both `h2xml` and `xml2py` are Python scrips. Therefore, the generation process can be automated in a simple generator script. This makes all steps reproducible, documents the used settings, and

makes the process robust towards evolutionary (smaller) changes in the C API. A largely simplified version is in the listing of Fig. 2.

```
# Skipped declaration of paths.
HEADER_FILE = 'lcms.h'
header_basename = os.path.splitext(HEADER_FILE)[0]

h2xml.main(['h2xml.py', header_path,
            '-c',
            '-o',
            '%s.xml' % header_basename])

SYMBOLS = ['cms.*', 'TYPE_.*', 'PT_.*', 'ic.*', 'LPcms.*', ...]
xml2py.main(['xml2py.py', '-kdfs',
             '-l%s' % library_path,
             '-o', module_path,
             '-r%s' % '|'.join(SYMBOLS),
             '%s.xml' % header_basename]
```

Figure 2: Essential parts of the code generator script.

Generated code should *never* be edited manually. As some modification will be necessary to achieve the desired functionality (see Sect. 4), automation becomes essential to yield reproducible results. Due to some shortcomings (see Sect. 4) of the generated code however, some editing was necessary. This modification has also been integrated into the generator script to fully remove the need of manual editing.

4 Refining the C API

In the current version of *Ctypes* in Python 2.5 it is not possible to add e. g. `__repr__()` or `__str__()` methods to data types. Also, code for loading the shared library in a platform independent way needs to be "patched" into the generated code. A function in the code generator reads the whole generated module `_lcms` and writes it back to the file system, and in the course replacing three lines from the beginning of the file with the code snippet from the listing in Fig. 3.

```
from _setup import *
import _setup

_libraries = {}
_libraries['/usr/lib/liblcms.so.1'] = _setup._init()
```

Figure 3: Lines to be patched into the generated module `_lcms`.

`_setup` (listing in Fig. 4) "monkey patches"[1] the class `ctypes.Structure` to include a `__repr__()` method (lines 4–10) for ease of use when representing wrapped objects for output. Furthermore, the loading of the shared library (DLL in Windows lingo) is abstracted to work in a platform independent way using the system's default search mechanism (lines 12–13).

4.1 Creating the Basic Wrapper

Further modifications are less invasive. For this, the C API is refined into a module `c_lcms`. This module imports *everything* from the `generated._lcms` and overrides or adds certain functionality

[1] A monkey patch is a way to extend or modify the runtime code of dynamic languages without altering the original source code: `http://en.wikipedia.org/wiki/Monkey_patch`

```
import ctypes
from ctypes.util import find_library

class Structure(ctypes.Structure):
    def __repr__(self):
        """Print fields of the object."""
        res = []
        for field in self._fields_:
            res.append('%s=%s' % (field[0], repr(getattr(self, field[0]))))
        return '%s(%s)' % (self.__class__.__name__, ', '.join(res))

def _init():
    return ctypes.cdll.LoadLibrary(find_library('lcms'))
```

Figure 4: Extract from module `_setup.py`.

individually (again through "monkey patching"). These are intended to make the C API a little bit easier to use through some helper functions, but mainly to make the new bindings more compatible with and similar to the official *SWIG* bindings (packaged together with *LittleCMS*). The wrapped C API can be used from Python (see Sect. 4.2). Although, it still requires closing, freeing or deleting from the code after use, and `c_lcms` objects/structures do not feature methods for operations. This shortcoming will be solved later.

4.2 `c_lcms` Example

The wrapped raw C API in Python behaves in exactly the same way, it is just implemented in Python syntax (listing in Fig. 5).

```
from c_lcms import *

def correctColour():
    inProfile  = cmsOpenProfileFromFile('HPSJTW.ICM', 'r')
    outProfile = cmsCreate_sRGBProfile()

    myTransform = cmsCreateTransform(inProfile, TYPE_RGB_8,
                                     outProfile, TYPE_RGB_8,
                                     INTENT_PERCEPTUAL, 0)

    for line in scanLines:
        # Skipped handling of buffers.
        cmsDoTransform(myTransform,
                       yourInBuffer,
                       yourOutBuffer,
                       numberOfPixelsPerScanLine)

    cmsDeleteTransform(myTransform)
    cmsCloseProfile(inProfile)
    cmsCloseProfile(outProfile)
```

Figure 5: Example using the basic API of the `c_lcms` module.

5 A Pythonic API

To create the usual pleasant "batteries included" feeling when working with code in Python, another module – `littlecms` – was manually created, implementing the *Façade Design Pattern.* From here

on we are moving away from the original C-like API. This high level object oriented Façade takes care of the internal handling of tedious and error prone operations. It also performs sanity checking and automatic detection for certain crucial parameters passed to the C API. This has drastically reduced problems with the low level nature of the underlying C library.

5.1 `littlecms` Example

Using `littlecms` the API is now object oriented (listing in Fig. 6) with a `doTransform()` method on the `myTransform` object. But there are a few more interesting benefits of this API:

- Automatic disposing of C API instances hidden inside the `Profile` and `Transform` classes.
- Largely reduced code size with an easily comprehensible structure.
- Redundant passing of information (e. g. the in- and output colour spaces) is determined within the `Transform` constructor from information available in the `Profile` objects.
- Uses *NumPy* [10] arrays for convenience in the buffers, rather than introducing further custom types. On these data array types and shapes can be automatically matched up.
- The number of pixels for each scan line placed in `yourInBuffer` can usually be detected automatically.
- Compatible with the often used *PIL* [11] library.
- Several sanity checks prevent clashes of erroneously passed buffer sizes, shapes, types, etc. that would otherwise result in a crashed or "hanging" process.

```
1  from littlecms import Profile, PT_RGB, Transform
2
3  def correctColour():
4      inProfile = Profile('HPSJTW.ICM')
5      outProfile = Profile(colourSpace=PT_RGB)
6      myTransform = Transform(inProfile, outProfile)
7
8      for line in scanLines:
9          # Skipped handling of buffers.
10         myTransform.doTransform(yourNumpyInBuffer, yourNumpyOutBuffer)
```

Figure 6: Example using the object oriented API of the `littlecms` module.

6 Conclusion

Binding pure C libraries to Python is not very difficult, and the skills can be mastered in a rather short time frame. If done right, these bindings can be quite robust even towards certain changes in the evolving C API without the need of very time consuming manual tracking of all changes. As with many projects for this, it is vital to be able to automate the "mechanical" processes: Beyond the outlined code generation in this paper, an important role comes to automated code integrity testing (here: using *PyUnit* [12]) as well as an API documentation (here: using *Epydoc* [13]).

Unfortunately, as *CtypesLib* is still work in progress, the whole process did not go as smoothly as described here. It was particularly important to match up working versions properly between GCCXML (which in itself is still in development) and *CtypesLib*. In this case a current GCCXML in version 0.9.0 (as available in Ubuntu Intrepid Ibex, 8.10) required a branch of *CtypesLib* that

needed to be checked out through the developer's Subversion repository. Furthermore, it was necessary to develop a fix for the code generator as it failed to generate code for `#define`d floating point constants. The patch has been reported to the author and is now in the source code repository. Also patching into the generated source code for overriding some features and manipulating the library loading code can be considered as being less than elegant.

Library wrapping as described in this paper was performed on version 1.16 of the *LittleCMS* library. While writing this paper the author has moved to the now stable version 1.17. Adapting the Python wrapper to this code base was a matter of about 15 minutes of work. The main task was fixing some unit tests due to rounding differences resulting from an improved numerical model within the library. The author of *LittleCMS* made a first preview of the upcoming version 2.0 (an almost complete rewrite) available recently. Adapting to that version took only about a good day of modifications, even though some substantial changes were made to the API. But even for this case only very little amounts of new code had to be written.

Overall, it is foreseeable that this type of library wrapping in the Python world will become more and more ubiquitous, as the tools for it mature. But already at the present time one does not have to fear the process. The time spent initially setting up the environment will be easily saved over all projects phases and iterations. It will be interesting to see *Ctypes* evolve to be able to interface to C++ libraries as well. Currently the developers of *Ctypes* and *Py++* (Thomas Heller and Roman Yakovenko) are evaluating potential extensions.

References

[1] *Official Python Documentation: Extending and Embedding the Python Interpreter*, Python Software Foundation.

[2] T. Heller, "Python Ctypes Project," http://starship.python.net/crew/theller/ctypes/, last accessed December 2008.

[3] M. Maria, "LittleCMS project," http://littlecms.com/, last accessed January 2009.

[4] D. M. Beazley and W. S. Fulton, "SWIG Project," http://www.swig.org/, last accessed December 2008.

[5] D. Abrahams and R. W. Grosse-Kunstleve, "Building Hybrid Systems with Boost.Python," http://www.boostpro.com/writing/bpl.html, March 2003, last accessed December 2008.

[6] D. Abrahams, "Boost.Python Project," http://www.boost.org/libs/python/, last accessed December 2008.

[7] S. Behnel, R. Bradshaw, and G. Ewing, "Cython Project," http://cython.org/, last accessed December 2008.

[8] R. Yakovenko, "Py++ Project," http://www.language-binding.net/pyplusplus/pyplusplus.html, last accessed December 2008.

[9] G. K. Kloss, "Source Code: Automatic C Library Wrapping – Ctypes from the Trenches," *The Python Papers Source Codes*, vol. 1, pp. –, January 2009, [Online available] http://ojs.pythonpapers.org/index.php/tppsc/issue/view/13.

[10] T. Oliphant, "NumPy Project," http://numpy.scipy.org/, last accessed December 2008.

[11] F. Lundh, "Python Imaging Library (PIL) Project," http://www.pythonware.com/products/pil/, last accessed December 2008.

[12] S. Purcell, "PyUnit Project," http://pyunit.sourceforge.net/, last accessed December 2008.

[13] E. Loper, "Epydoc Project," http://epydoc.sourceforge.net/, last accessed December 2008.