

Source Code: 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*, with the help of the matching source code generator from *CtypesLib*.

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

1 Overview

One of the grand fundamentals in software engineering is to use the tools that are best suited for a job, and not to decide on an implementation prematurely. That is often easier said than done, in the light of some complimentary requirements (e.g. rapid/easy implementation vs. the need for speed of execution or vs. low level access to hardware). The traditional way 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 *Ctypes* [1], which is by default part of Python since version 2.5.

As an example the creation of a wrapper for the *LittleCMS* colour management library [2] is outlined. The library offers excellent features, and ships with “official” Python bindings (using *SWIG* [3]), 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.

In this case the C library is facilitated from Python through code generation. The generated code is refined in an API module to meet the desired functionality of the wrapper. As the library is anything but “Pythonic,” an object oriented wrapper API for the library that features “qualities we love” is built on top.

A more complete description of the wrapping process can be found in the corresponding article in The Python Papers [4].

2 Requirements

The work presented in this paper is based on wrapping the *LittleCMS* colour management library in version 1.16. It has also been used together with version 1.17, which required only the most minor tweaks, mainly on unit tests.

For the development itself, the code generator from the *CtypesLib* project is needed. For parsing the library’s header file it uses *GCCXML*, the *GCC* compiler’s own parser that produces an XML representation of the code’s parse tree. In most current Linux distributions now version 0.9.0 of *GCCXML* is available. This version requires either the “`ctypeslib-gccxml-0.9`” branch of *CtypesLib* or a recent snapshot of the development branch.

Finally, for the execution of the wrapper libraries the *Ctypes* as well as the *NumPy*¹ modules are required. Implementing the wrappers differently, one can avoid using *NumPy* at the expense of a largely sacrificed convenience when operating on (larger) array structures. The *Python Imaging Library (PIL)*² may be a suitable companion for experimenting with images in using the presented bindings.

3 The Code

This code is free software: you can redistribute and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

3.1 Code Generator

The author of *Ctypes* is developing *CtypesLib*. It contains a code generator in the modules `ctypeslib.h2xml` and `ctypeslib.xml2py`. These can be called manually, or from a generator script (Fig. 1) with the proper parameters for the task to automate the process. The header is parsed (lines 16–19), and a Python binding module is generated into a module `_lcms` (lines 21–25). For some “deeper” modifications code needs to be evaluated *before* the bindings are defined. For this purpose the generator patches (lines 27–36) the generated module `_lcms` by *injecting* the code from `_setup.py` (Fig. 2).

Some excerpts of the quite extensive generated code from `_lcms.py` is shown in Fig. 3. Lines 3–7 have been “patched” into it by the code generator in place of three original lines. Some general (lines 9–11) as well as some library specific simple data types are defined (lines 13–14). Constants (explicit and computed from C pre-processor macros) are as well defined (lines 16–17). In the following of Fig. 3 are several more complex type definitions from C structures. These are mostly different data containers that can be used as buffer types (`icUInt8Array` or `cmsCIEXYZ`)

¹<http://numpy.scipy.org/>

²<http://www.pythonware.com/products/pil/>

for the `cmsDoTransform()` function after casting to `LPVOID` (see lines 13–16 in Fig. 9). `cmsCIEXYZTRIPLE` in contrast shows a nested data type as it would be used e. g. to describe the absolute colourimetric values of a device’s primary colours (as a camera).

Fig. 4 shows the generated signature definitions for calling wrapped C functions. All C functions used in the sample application in Fig. 9 are shown. Basically a callable Python object is bound to an exposed stub in the library. The attribute `argtypes` describes a list of the calling parameter types, whereas `restype` describes the type of the returned value.

3.2 The API

3.2.1 `c_lcms.py`

Some features of the original *SWIG* API have not been mapped identically through the code generation. These issues and some helpers are taken care of in the end user C binding API `c_lcms` as shown in excerpts in Fig. 5. The first lines import the whole generated name space `_lcms`. A pre-processor macro’s functionality is mapped through a function (lines 4–6), some missing data types are created (lines 12–15) and equipped with a nice output representation (lines 8–10). Also some *SWIG* naming tweak is emulated (line 17).

Colour space type descriptors are used in a human readable verbatim form (as a string), an integer constant from `lcms.h` as well as one from the required `icc34.h`. These corresponding constants are stored in an easy to handle Python dictionary containing `ColourSpace` objects, which in turn contain the different representations (lines 35–42). `ColourSpace` gets its functionality from `PropertyContainer` (lines 19–29).

3.2.2 `littlecms.py`

`littlecms.py` (see Fig. 6) finally contains the object oriented and Pythonic parts of the end user API. First of all a handler that is called by the native library on all errors is defined and assigned (lines 6–13). The error handler also raises an `LcmsException`.

The `Profile` class loads profiles from the file system, or creates embedded ones from the built-in library in the constructor (lines 21–28), which it also removes whenever the `Profile` object is discarded (lines 30–32). Furthermore, it reveals the profile `name` as well as `colourSpace` attribute information (lines 34–40) through object introspection and use of the `ColourSpace` helper structures defined in `c_lcms`.

In the `Transform` class (Fig. 7) the two profiles (input and output profile) are jointly used to transform colour tuples between colour representations. The constructor is particularly helpful in using the library. Besides several sanity checks for robustness it implements an automatic detection of the involved colour spaces through the before mentioned profile introspection (lines 64–70). The creation of the actual (internal) transformation structure is located in the function

`_createTransform()` (lines 78–89). The reason is, that on every update of an attribute of a `Transform` instance this transform structure needs to be disposed and replaced by a new one. Management of this functionality is hidden through use of the `property` decorator together with the `operator.attrgetter` function (Fig. 8, lines 91–105). This way operations are exposed to the end user as if they were simple attributes of an object that can be assigned or retrieved without the need of any helper methods. Finally, for the `doTransform()` method (lines 107–121) in absence of a set `destinationBuffer` a buffer of a suitable type, size and shape will be created. This way the `doTransform()` method can be used in the way of an assignment operation, returning a suitable `numpy` array. Alternatively, it can be called with input and output buffers, or even with an output buffer which is identical with the input buffer. In the latter case an in-place transformation will be performed, overwriting the input data with the transformed colour representation.

3.3 Examples

Finally two examples are presented. One is using the direct C wrapped user space API from `c_lcms` that is largely compatible to the official *SWIG* bindings (Fig. 9). A scanned image is converted from the device specific colour space from a HP ScanJet scanner as characterised in the file `HPSJTW.ICM`, to the standardised sRGB display colour space using a built in profile (lines 4, 5). The transformation is performed line-by-line, as pixel rows in images are often padded to multiples of certain sizes (lines 11-16). The number of pixels (colour tuples) per pixel row must be specified (line 16). In case the buffers are `numpy` arrays, the buffer must be passed e.g. as `yourInBuffer.ctypes`, or in case of a *Ctypes* buffer using `ctypes.byref(yourInBuffer)` (lines 14, 15). Due to the fact that whole pixel rows can be transformed within the native C library “in one go,” the performance is very good. Finally, *LittleCMS* structures that were created must be manually freed again (lines 18–20).

The same task using `littlecms` simplifies the handling even for this simple example quite significantly (see Fig. 10). The buffers are native `numpy` arrays, and need no specific calling conventions. If the buffers consist of a two dimensional array (array of tuples for each pixel), then the number of pixels for the buffer conversion is automatically detected. As for example also the `PIL` module supports handling of image data as `numpy` arrays, the usage becomes quite simple. As `numpy` arrays are internally implemented in a pure C library, no speed degradation should be noticeable.

References

- [1] T. Heller, “Python Ctypes Project,” <http://starship.python.net/crew/theller/ctypes/>, last accessed December 2008.
- [2] M. Maria, “LittleCMS project,” <http://littlecms.com/>, last accessed December 2008.
- [3] D. M. Beazley and W. S. Fulton, “SWIG Project,” <http://www.swig.org/>, last accessed December 2008.
- [4] G. K. Kloss, “Automatic C Library Wrapping – Ctypes from the Trenches,” *The Python Papers*, vol. 3, no. 3, January 2008, [Online available] <http://ojs.pythonpapers.org/index.php/tpp/issue/view/10>.

```

1 from ctypeslib import h2xml, xml2py

3 HEADER_FILE = 'lcms.h'
4 SYMBOLS = ['cms.*', 'TYPE.*', 'PT.*', 'ic.*', 'LPcms.*', 'LCMS.*',
5           'lcms.*', 'PERCEPTUAL_BLACK.*', 'INTENT.*', 'GAMMA.*']
6 # Skipped some constants.

8 GENERATOR_PATCH = """
9 from _setup import *
10 import _setup

12 _libraries = {}
13 _libraries['/usr/lib/liblcms.so.1'] = _setup._init()
14 """

16 def parse_header(header):
17     # [Skipped "path magic".]
18     h2xml.main(['h2xml.py', header_path, '-c', '-o',
19              '%s.xml' % header_basename])

21 def generate_code(header):
22     # [Skipped "path magic".]
23     xml2py.main(['xml2py.py', '-kdfs', '-l%s' % LIBRARY_PATH,
24               '-o', module_path, '-r%s' % '|'.join(SYMBOLS),
25               '%s.xml' % header_basename])

27 def patch_module(header):
28     # [Skipped "path magic".]
29     fd = open(module_path)
30     lines = fd.readlines()
31     fd.close()
32     fd = open(module_path, 'w')
33     fd.write(lines[0])
34     fd.write(GENERATOR_PATCH)
35     fd.writelines(lines[4:])
36     fd.close()

38 def main():
39     parse_header(HEADER_FILE)
40     generate_code(HEADER_FILE)
41     patch_module(HEADER_FILE)

```

Figure 1: Essential parts of the code generator.

```
1 import ctypes
2 from ctypes.util import find_library

4 # One of Gary Bishop's ctypes tricks:
5 # http://wwwx.cs.unc.edu/~gb/wp/blog/2007/02/11/ctypes-tricks/
6 # Hack the ctypes.Structure class to include printing the fields.
7 class Structure(ctypes.Structure):
8     def __repr__(self):
9         """Print fields of the object."""
10        res = []
11        for field in self._fields_:
12            res.append('%s=%s' % (field[0], repr(getattr(self, field[0]))))
13        return '%s(%s)' % (self.__class__.__name__, ', '.join(res))

15    @classmethod
16    def from_param(cls, obj):
17        """Magically construct from a tuple."""
18        if isinstance(obj, cls):
19            return obj
20        if isinstance(obj, tuple):
21            return cls(*obj)
22        raise TypeError

24 def _init():
25     """Hunts down and loads the shared library."""
26     return ctypes.cdll.LoadLibrary(find_library('lcms'))
```

Figure 2: Essential parts of the `_setup` module used for “patching” the generated code in `_lcms.py`.

```
1 from ctypes import *
3 from _setup import *
4 import _setup
6 _libraries = {}
7 _libraries['liblcms.so.1'] = _setup._init()
9 STRING = c_char_p
10 DWORD = c_ulong
11 LPVOID = c_void_p
13 LCMSHANDLE = c_void_p
14 cmsHPROFILE = LCMSHANDLE
16 TYPE_RGB_8 = 262169 # Variable c_int '262169'
17 INTENT_PERCEPTUAL = 0 # Variable c_int '0'
19 class icUInt8Array(Structure):
20     pass
21 u_int8_t = c_ubyte
22 icUInt8Number = u_int8_t
23 icUInt8Array._fields_ = [
24     ('data', icUInt8Number * 1),
25 ]
27 class cmsCIEXYZ(Structure):
28     pass
29 cmsCIEXYZ._pack_ = 4
30 cmsCIEXYZ._fields_ = [
31     ('X', c_double),
32     ('Y', c_double),
33     ('Z', c_double),
34 ]
36 class cmsCIEXYZTRIPLE(Structure):
37     pass
38 cmsCIEXYZTRIPLE._fields_ = [
39     ('Red', cmsCIEXYZ),
40     ('Green', cmsCIEXYZ),
41     ('Blue', cmsCIEXYZ),
42 ]
```

Figure 3: Edited excerpts from `_lcms.py`.

```
44 cmsOpenProfileFromFile = _libraries['liblcms.so.1'].cmsOpenProfileFromFile
45 cmsOpenProfileFromFile.restype = cmsHPROFILE
46 cmsOpenProfileFromFile.argtypes = [STRING, STRING]

48 cmsCreate_sRGBProfile = _libraries['liblcms.so.1'].cmsCreate_sRGBProfile
49 cmsCreate_sRGBProfile.restype = cmsHPROFILE
50 cmsCreate_sRGBProfile.argtypes = []

52 cmsCreateTransform = _libraries['liblcms.so.1'].cmsCreateTransform
53 cmsCreateTransform.restype = cmsHTRANSFORM
54 cmsCreateTransform.argtypes = [cmsHPROFILE, DWORD,
55                                cmsHPROFILE, DWORD,
56                                c_int, DWORD]

58 cmsDoTransform = _libraries['liblcms.so.1'].cmsDoTransform
59 cmsDoTransform.restype = None
60 cmsDoTransform.argtypes = [cmsHTRANSFORM, LPVOID, LPVOID, c_uint]

62 cmsDeleteTransform = _libraries['liblcms.so.1'].cmsDeleteTransform
63 cmsDeleteTransform.restype = None
64 cmsDeleteTransform.argtypes = [cmsHTRANSFORM]

66 cmsCloseProfile = _libraries['liblcms.so.1'].cmsCloseProfile
67 cmsCloseProfile.restype = BOOL
68 cmsCloseProfile.argtypes = [cmsHPROFILE]
```

Figure 4: `_lcms.py` continued: Edited excerpts of function definitions.

```

1 import ctypes
2 from generated._lcms import *

4 # A flag generating C macro reimplemented as a function.
5 def cmsFLAGS_GRIDPOINTS(n):
6     return (n & 0xFF) << 16

8 # SWIG wrapper backwards compatibility definitions.
9 def __array_repr__(self):
10    return '%s(%s)' % (self.__class__.__name__, [x for x in self])

12 COLORB = ctypes.c_ubyte * 3
13 COLORB.__repr__ = __array_repr
14 COLORW = ctypes.c_uint16 * 3
15 COLORW.__repr__ = __array_repr

17 cmsSaveProfile = _cmsSaveProfile

19 class PropertyContainer(object):
20    """Container class for simple property objects."""
21    def __init__(self, **attributes):
22        self.__dict__ = attributes

24    def __repr__(self):
25        """Print fields of the object."""
26        res = []
27        for attribute, value in self.__dict__.items():
28            res.append('%s=%s' % (attribute, value.__repr__()))
29        return '%s(%s)' % (self.__class__.__name__, ', '.join(res))

31 class ColourSpace(PropertyContainer):
32    """A colour space descriptor."""

34 # Colour space type descriptors for lcms.h and icc34.h.
35 colourTypeFromName = {
36     'GRAY': ColourSpace(name='GRAY', lcms=PT_GRAY, ICC=icSigGrayData),
37     'RGB': ColourSpace(name='RGB', lcms=PT_RGB, ICC=icSigRgbData),
38     'CMYK': ColourSpace(name='CMYK', lcms=PT_CMYK, ICC=icSigCmykData),
39     # [Some snipped out.]
40     'XYZ': ColourSpace(name='XYZ', lcms=PT_XYZ, ICC=icSigXYZData),
41     'Lab': ColourSpace(name='Lab', lcms=PT_Lab, ICC=icSigLabData),
42 }

```

Figure 5: Edited extract from `c_lcms.py`.

```
1 from operator import attrgetter
2 import numpy
3 import ctypes
4 from c_lcms import *

6 class LcmsException(Exception):
7     """Indicates that an Exception in the Lcms module has occurred."""

9 def __lcmsErrorHandler(errorCode, errorText):
10     """Error handler called by liblcms on errors."""
11     # [Error level determination skipped.]
12     message = '%s: %s!' % (errorCode, errorText)
13     raise LcmsException(message)

15 lcmsErrorHandler = cmsErrorHandlerFunction(__lcmsErrorHandler)
16 cmsErrorAction(LCMS_ERROR_SHOW)
17 cmsSetErrorHandler(lcmsErrorHandler)

19 class Profile(object):
20     """Profile handling class."""
21     def __init__(self, fileName=None, colourSpace=None):
22         self._profile = None
23         if fileName != None:
24             self._profile = cmsOpenProfileFromFile(fileName, 'r')
25         elif colourSpace != None:
26             # [Built in profile creation skipped.]
27         else:
28             raise LcmsException('Unknown profile type to create.')

30     def __del__(self):
31         if self._profile:
32             cmsCloseProfile(self._profile)

34     @property
35     def name(self):
36         return cmsTakeProductName(self._profile)

38     @property
39     def colourSpace(self):
40         return colourTypeFromICC[cmsGetColorSpace(self._profile)]
```

Figure 6: Edited extract from `littlecms.py`. Error handler and `Profile` class.

```

42 # [Some internal helper snipped.]
44 class Transform(object):
45     """Transformation handling class."""
47     def __init__(self, inputProfile, outputProfile,
48                 renderingIntent=INTENT_PERCEPTUAL,
49                 transformationFlags=cmsFLAGS_NOTPRECALC,
50                 inputDepth=8, outputDepth=8,
51                 specialInputFormat=None, specialOutputFormat=None):
52         self._myTransform = None
53         self._inputProfile = inputProfile
54         self._outputProfile = outputProfile
55         self._renderingIntent = renderingIntent
56         self._transformationFlags = transformationFlags
57         # [Sanity check for allowed input and output bit depths snipped.]
58         self.inputDepth = inputDepth
59         self.outputDepth = outputDepth
60         # [Some further sanity checks snipped.]
61         self.specialInputFormat = specialInputFormat
62         self.specialOutputFormat = specialOutputFormat
64         # Detect the input/output format.
65         self._inputFormat = _getColourType(self._inputProfile,
66                                           self.inputDepth,
67                                           self.specialInputFormat)
68         self._outputFormat = _getColourType(self._outputProfile,
69                                             self.outputDepth,
70                                             self.specialOutputFormat)
72         self._createTransform()
74     def __del__(self):
75         if self._myTransform:
76             cmsDeleteTransform(self._myTransform)
78     def _createTransform(self):
79         if self._myTransform:
80             cmsDeleteTransform(self._myTransform)
81             self._myTransform = None
82         self._myTransform = cmsCreateTransform(self._inputProfile._profile,
83                                             self._inputFormat,
84                                             self._outputProfile._profile,
85                                             self._outputFormat,
86                                             self._renderingIntent,
87                                             self._transformationFlags)
88         if self._myTransform == None:
89             raise LcmsException('Error creating transform.')

```

Figure 7: `littlecms.py` continued: Edited extract from `littlecms.py`. Transform class creation and disposal.

```
91     def _setInputProfile(self, aProfile):
92         self._inputProfile = aProfile
93         self._createTransform()
94     inputProfile = property(attrgetter('_inputProfile'), _setInputProfile)

96     def _setOutputProfile(self, aProfile):
97         self._outputProfile = aProfile
98         self._createTransform()
99     outputProfile = property(attrgetter('_outputProfile'), _setOutputProfile)

101    def _setTransformationFlags(self, theTransformationFlags):
102        self._transformationFlags = theTransformationFlags
103        self._createTransform()
104    transformationFlags = property(attrgetter('_transformationFlags'),
105                                   _setTransformationFlags)

107    def doTransform(self, sourceBuffer, destinationBuffer=None,
108                   numberTuples=None):
109        if numberTuples == None:
110            numberTuples = len(sourceBuffer)
111        if destinationBuffer == None:
112            # [depth_type determination skipped.]
113            destinationBuffer = numpy.zeros(sourceBuffer.shape,
114                                           dtype=depth_type)

116        # [Sanity checks for buffers compatibility skipped.]
117        cmsDoTransform(self._myTransform,
118                      sourceBuffer.ctypes,
119                      destinationBuffer.ctypes,
120                      numberTuples)
121    return destinationBuffer
```

Figure 8: `littlecms.py` continued: Edited extract from `littlecms.py`. Alterations of Transform object and `doTransform()` method.

```
1 from c_lcms import *
3 def correctColour():
4     inProfile = cmsOpenProfileFromFile('HPSJTW.ICM', 'r')
5     outProfile = cmsCreate_sRGBProfile()
7     myTransform = cmsCreateTransform(inProfile, TYPE_RGB_8,
8                                     outProfile, TYPE_RGB_8,
9                                     INTENT_PERCEPTUAL, 0)
11    for line in scanLines:
12        # Skipped handling of buffers.
13        cmsDoTransform(myTransform,
14                      ctypes.byref(yourInBuffer),
15                      ctypes.byref(yourOutBuffer),
16                      numberOfPixelsPerScanLine)
18    cmsDeleteTransform(myTransform)
19    cmsCloseProfile(inProfile)
20    cmsCloseProfile(outProfile)
```

Figure 9: Example using the basic API of the `c_lcms` module.

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

Figure 10: Example using the object oriented API of the `littlecms` module.