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 new 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\textsuperscript{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)\textsuperscript{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) 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

\textsuperscript{1}http://numpy.scipy.org/
\textsuperscript{2}http://www.pythonware.com/products/pil/
from ctypeslib import h2xml, xml2py

HEADER_FILE = '\lcms.h'
SYMBOLS = ['cms.*', 'TYPE.*', 'PT.*', 'ic.*', 'LPCms.*', 'LCMS.*',
'\lcms.*', 'PERCEPTUAL_BLACK.*', 'INTENT.*', 'GAMMA.*']

# Skipped some constants.

GENERATOR_PATCH = ""
from _setup import *
import _setup

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

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

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

def patch_module(header):
    # [Skipped "path magic".]
    fd = open(module_path)
    lines = fd.readlines()
    fd.close()
    fd = open(module_path, 'w')
    fd.write(lines[0])
    fd.write(GENERATOR_PATCH)
    fd.writelines(lines[4:])
    fd.close()

def main():
    parse_header(HEADER_FILE)
    generate_code(HEADER_FILE)
    patch_module(HEADER_FILE)

Figure 1: Essential parts of the code generator.

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 a 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
import ctypes

from ctypes.util import find_library

# One of Gary Bishop's ctypes tricks:
# Hack the ctypes.Structure class to include printing the fields.

class Structure(ctypes.Structure):
    # Hack the ctypes.Structure class to include printing the fields.
    # Hack the ctypes.Structure class to include printing the fields.

    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))

    @classmethod
    def from_param(cls, obj):
        """Magically construct from a tuple.""
        if isinstance(obj, cls):
            return obj
        if isinstance(obj, tuple):
            return cls(*obj)
        raise TypeError

    def __init__(self):
        """Hunts down and loads the shared library.""

        return ctypes.cdll.LoadLibrary(find_library('lcms'))

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

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 cre-
ation 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 trans-
formation structure needs to be disposed and replaced by a new one. Management of this functionali-
ty 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 HP3JTW.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).
from ctypes import *
from _setup import *
import _setup

_libraries = {}
_libraries['liblcms.so.1'] = _setup._init()

STRING = c_char_p
DWORD = c_ulong
LPVOID = c_void_p

LCMSHANDLE = c_void_p
.cmsHPROFILE = LCMSHANDLE

TYPE_RGB_8 = 262169  # Variable c_int '262169'
INTENT_PERCEPTUAL = 0  # Variable c_int '0'

class icUInt8Array(Structure):
    pass
    u_int8_t = c_ubyte
    icUInt8Number = u_int8_t
    icUInt8Array._fields_ = [
        ('data', icUInt8Number * 1),
    ]

class cmsCIEXYZ(Structure):
    pass
cmsCIEXYZ._pack_ = 4
cmsCIEXYZ._fields_ = [
    ('X', c_double),
    ('Y', c_double),
    ('Z', c_double),
]

class cmsCIEXYZTRIPLE(Structure):
    pass
cmsCIEXYZTRIPLE._fields_ = [
    ('Red', cmsCIEXYZ),
    ('Green', cmsCIEXYZ),
    ('Blue', cmsCIEXYZ),
]

Figure 3: Edited excerpts from _lcms.py.

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

References


import ctypes

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

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

COLORB = ctypes.c_ubyte * 3
COLORB.__repr__ = __array_repr
COLORW = ctypes.c_uint16 * 3
COLORW.__repr__ = __array_repr

cmsSaveProfile = _cmsSaveProfile

class PropertyContainer(object):
    """Container class for simple property objects.""

def __init__(self, **attributes):
    self.__dict__ = attributes

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

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

# Colour space type descriptors for lcms.h and icc34.h.
colourTypeFromName = {
    'GRAY': ColourSpace(name='GRAY', lcms=PT_GRAY, ICC=icSigGrayData),
    'RGB': ColourSpace(name='RGB', lcms=PT_RGB, ICC=icSigRgbData),
    'CMYK': ColourSpace(name='CMYK', lcms=PT_CMYK, ICC=icSigCmykData),
    'XYZ': ColourSpace(name='XYZ', lcms=PT_XYZ, ICC=icSigXYZData),
    'Lab': ColourSpace(name='Lab', lcms=PT_Lab, ICC=icSigLabData),
}

Figure 5: Edited extract from c_lcms.py.
from operator import attrgetter
import numpy
import ctypes
from c.lcms import *

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

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

lcmsErrorHandler = cmsErrorHandlerFunction(__lcmsErrorHandler)
cmsErrorAction(LCMS.ERROR_SHOW)
cmsSetErrorHandler(lcmsErrorHandler)

class Profile(object):
    """Profile handling class.""
    def __init__(self, fileName=None, colourSpace=None):
        self._profile = None
        if fileName != None:
            self._profile = cmsOpenProfileFromFile(fileName, 'r')
        elif colourSpace != None:
            # [Built in profile creation skipped.]
        else:
            raise LcmsException('Unknown profile type to create.')
    def __del__(self):
        if self._profile:
            cmsCloseProfile(self._profile)
    @property
def name(self):
        return cmsTakeProductName(self._profile)
    @property
def colourSpace(self):
        return colourTypeFromICC[cmsGetColorSpace(self._profile)]

Figure 6: Edited extract from littlecms.py. Error handler and Profile class.
class Transform(object):
    """Transformation handling class."""
    def __init__(self, inputProfile, outputProfile, 
        renderingIntent=INTENT_PERCEPTUAL, 
        transformationFlags=cmsFLAGS_NOTPRECALC, 
        inputDepth=8, outputDepth=8, 
        specialInputFormat=None, specialOutputFormat=None):
        self._myTransform = None
        self._inputProfile = inputProfile
        self._outputProfile = outputProfile
        self._renderingIntent = renderingIntent
        self._transformationFlags = transformationFlags
        # [Safety check for allowed input and output bit depths snipped.]
        self.inputDepth = inputDepth
        self.outputDepth = outputDepth
        # [Some further sanity checks snipped.]
        self.specialInputFormat = specialInputFormat
        self.specialOutputFormat = specialOutputFormat
        # Detect the input/output format.
        self._inputFormat = _getColourType(self._inputProfile, 
            self.inputDepth, 
            self.specialInputFormat)
        self._outputFormat = _getColourType(self._outputProfile, 
            self.outputDepth, 
            self.specialOutputFormat)
        self._createTransform()

    def __del__(self):
        if self._myTransform:
            cmsDeleteTransform(self._myTransform)

    def _createTransform(self):
        if self._myTransform:
            cmsDeleteTransform(self._myTransform)
        self._myTransform = None
        self._myTransform = cmsCreateTransform(self._inputProfile._profile, 
            self._inputFormat, 
            self._outputProfile._profile, 
            self._outputFormat, 
            self._renderingIntent, 
            self._transformationFlags)
        if self._myTransform == None:
            raise LcmsException('Error creating transform.')

Figure 7: littlecms.py continued: Edited extract from littlecms.py. Transform class creation and disposal.
def _setInputProfile(self, aProfile):
    self._inputProfile = aProfile
    self._createTransform()
    inputProfile = property(attrgetter('_inputProfile'), _setInputProfile)

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

def _setTransformationFlags(self, theTransformationFlags):
    self._transformationFlags = theTransformationFlags
    self._createTransform()
    transformationFlags = property(attrgetter('_transformationFlags'), _setTransformationFlags)

def doTransform(self, sourceBuffer, destinationBuffer=None, numberTuples=None):
    if numberTuples == None:
        numberTuples = len(sourceBuffer)
    if destinationBuffer == None:
        destinationBuffer = numpy.zeros(sourceBuffer.shape,
        dtype=depth_type)
    cmsDoTransform(self._myTransform,
    sourceBuffer.ctypes,
    destinationBuffer.ctypes,
    numberOfPixelsPerScanLine)
    return destinationBuffer

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

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,
        ctypes.byref(yourInBuffer),
        ctypes.byref(yourOutBuffer),
        numberOfPixelsPerScanLine)
        cmsDeleteTransform(myTransform)
    cmsCloseProfile(inProfile)
    cmsCloseProfile(outProfile)

Figure 9: Example using the basic API of the c._lcms module.
from littlecms import Profile, PT_RGB, Transform

def correctColour():
inProfile = Profile('HPSJTW.ICM')
outProfile = Profile(colourSpace=PT_RGB)
myTransform = Transform(inProfile, outProfile)

    for line in scanLines:
        # Skipped handling of buffers.
        myTransform.doTransform(yourNumpyInBuffer, yourNumpyOutBuffer)

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