NiceLib Documentation

Release 0.6

Nate Bogdanowicz

Mar 01, 2019

Contents

NiceLib is a package for rapidly developing "nice" Python bindings to C libraries, using `cffi`.

It lets you turn a C function like this

```
int my_c_function(int arg1, float arg2, uint* out_arr, int out_arr_size, int␣
↪reserved);
```

into a Python function that can be called like this

```
out_arr = my_c_function(arg1, arg2)
```

just by defining a signature like this

```
class MyLib(NiceLib):
    ...
    my_c_function = Sig('in', 'in', 'arr', 'len', 'ignore')
```

while giving you easy error handling and more.

CHAPTER 1

Installing

NiceLib is available on PyPI:

```
$ pip install nicelib
```

If you would like to use the development version, download and extract a zip of the source from our GitHub page or clone it using git. Now install:

```
$ cd /path/to/NiceLib
$ pip install .
```

CHAPTER 2

Feedback / Contributing

For contributing, reporting issues, and providing feedback, see our GitHub page. Feedback is greatly appreciated!

CHAPTER 3

Introduction

NiceLib essentially consists of two layers:

- Tools for directly using C headers to generate an importable Python module
- An interface for quickly and cleanly defining a Pythonic mid-level binding

Mid-level means that functions take *input* arguments, return *output* arguments, and raise Exceptions on error. NiceLib tries to simplify commonly-seen C idioms, like using user-created buffers for returning output strings. Take, for example, this toy binding for the NIDAQmx library:

```
from nicelib import load_lib, NiceLib, Sig

class NiceNI(NiceLib):
    _info_ = load_lib('ni')
    _prefix_ = 'DAQmx'

    GetSysDevNames = Sig('buf', 'len')
```

Now, we can simply call `NiceNI.GetSysDevName(code)`, which is equivalent to the following:

```
lib_info = load_lib('ni')
ffi, lib = lib_info._ffi, lib_info.lib

def GetSysDevNames():
    buflen = 512
    buf = ffi.new('char[]', buflen)
    ret = lib.DAQmxGetSysDevNames(buf, buflen)
    return ffi.string(buf), ret
```

Many of our C library's functions may use status codes as their return value. To automatically check the return code and raise an exception if warranted, we can use `_ret_`:

```
from nicelib import RetHandler

# Define a new return handler
```

(continues on next page)

(continued from previous page)

```
@RetHandler(num_retvals=0)
def daq_errorcheck(ret):
    if ret != 0:
        raise DAQError(ret)

class NiceNI(NiceLib):
    _ret_ = daq_errorcheck
    [...]
```

Often a C library will have method-like functions, each of which take a handle as its first argument. These translate naturally into Python objects, obviating you of the need to pass the handle all the time. For example, here's a partial definition of a Task object, again wrapping NIDAQmx:

```
class NiceNI(NiceLib):
    [...]
    CreateTask = Sig('in', 'out')

    class Task(NiceObject):
        _init_ = 'CreateTask'

        StartTask = Sig('in')
        ReadAnalogScalarF64 = Sig('in', 'in', 'out', 'ignore')
```

For both of these function signatures, the first `'in'` argument is the Task handle. We can then use the class to make and use `Task` objects:

```
task = NiceNI.Task('myTask')
task.StartTask()
```

By specifying `_init_` when defining `Task`, the string `'myTask'` is passed to `NiceNI.CreateTask`, whose return value (a task handle) is stored in the `Task` instance.

3.1 Automatically Processing Headers

The awesome `cffi` package greatly improves wrapping C libraries in Python by allowing you to load header files directly, instead of manually churning out mind-numbing `ctypes` boilerplate. However it's not perfect—in particular, it makes a fairly limited attempt at preprocessing header files, meaning that in many cases you'll have to preprocess them yourself, either manually or by running e.g. the gcc preprocessor. NiceLib provides preprocessing facilities that aim to allow you to use unmodified header files to generate a "compiled" ffi module, without requiring a C compiler.

NiceLib's preprocessor has basic support for both object-like and simple function-like macros, which it translates into equivalent[1] portable Python source code. For example

```
#define CONST_VAL       (1 << 4) | (1 << 1)
#define LTZ(val)        ((val) < 0)
```

becomes the Python code:

```
CONST_VAL = (1 << 4) | (1 << 1)
def LTZ(val):
    return (val) < 0
```

[1] Note that, due to the nature of the C preprocessor, this generated code cannot always be truly equivalent. However, in the overwhelming majority of cases, the macros defined in library header files are quite simple.

The preprocessor also supports conditionals (`#ifdef` and friends), `#includes`, and platform-specific predefined macros (like `__linux__`, `__WIN64`, and `__x86_64`).

Currently, `#pragma once` is supported, but other `#pragma` directives are ignored.

CHAPTER 4

User Guide

4.1 Creating Low-Level Bindings

Before you can write mid-level bindings to your library, you first need to create low-level bindings via `cffi`. Normally this would involve preprocessing your headers via `gcc -e`, followed by additional manual cleaning. Then you'd have to distribute this header with your software, with possible copyright implications.

Ideally, you wouldn't have to distribute a header at all since the user has the library and its headers installed already. NiceLib's goal is to make this possible.

If all goes well, you need to do only one thing: write a build module. For a library named `foo`, name your module file `_build_foo.py`, and put it in the same directory that your wrapper will be in. This build file contains info about where to find the shared lib and its headers on different platforms. The build module for a Windows-only lib might look like this:

```
# _build_foo.py
from nicelib import build_lib

header_info = {
    'win*': {
        'path': (
            r"{PROGRAMFILES}\Vendor\Product",
            r"{PROGRAMFILES(X86)}\Vendor\Product",
        ),
        'header': 'foo.h'
    },
}

lib_names = {'win*': 'foo'}


def build():
    build_lib(header_info, lib_names, '_foolib', __file__)
```

You then call `load_lib('foo', __package__)` in your wrapper file to load the `LibInfo` object. This uses the `_foolib` submodule if it exists. If it doesn't exist yet, `load_lib()` tries to build it by calling the `build()` function in `_build_foo`. This calls `build_lib()`, which searches for `foo.dll` in the system path and looks for `foo.h` in both of the vendor-specific directories given above. If it finds them successfully, it then processes the header so that `cffi` can understand it.

The two main challenges in writing a build file are locating the headers/libs and ensuring that the headers are processed succesfully.

4.1.1 Locating Headers and Libraries

Both the `header_info` and `lib_name` arguments to `build_lib` can be a dict that maps from a platform to the corresponding path or name, allowing cross-platform support. The platform specifiers (`'win*'` in the example above) are checked against `sys.platform` to find which platform-specific paths and filenames to try, using pattern globbing if given. You may also discriminate between 32- and 64-bit systems by appending a colon and then the bitness, e.g. `'win*:32'`. If you want the platforms to be checked in a specific order, for example if you want to specify a default fallthrough option, you can use an `ordereddict` instead of a `dict`.

For `lib_name`, each platform-specific value is a string or tuple of strings. Each string is the name of a library, without any prefix like `lib` or or suffix like `.so` or `.dll`. This is the form used by `ctypes.util.find_library`. If you use a tuple of names, NiceLib will look for each library in turn, using the first one it finds. This is useful if the library could have any of a few different names.

For `header_info`, the each platform-specific value is a dict with the following keys:

'header' A string or tuple of strings which are the names/paths of all the headers to include.

'path' Optional. A tuple of base directories where the headers may be found. Each header is searched for in each directory until it is found. The directories are specified as strings.

Path strings can contain environment variables in the form `'{VAR_NAME}'` as shown in the example above. If a variable is not contained in `os.environ`, the whole string is left unsubstituted.

Paths can be relative or absolute. Relative paths are relative to the directory given via the `filedir` parameter.

4.1.2 Processing Headers

Processing headers can be one of the trickier aspects of using NiceLib, especially if you're new to it. But don't be discouraged, there are tools designed to help you out.

If `build_lib()` doesn't succeed on your first try, you'll have to do a bit of sleuthing. First of all, if it looks like NiceLib is processing (or failing to find) a bunch of headers that you don't need, you can tell `build_lib()` to ignore them. If a header you're processing includes `<windows.h>`, for instance, header processing will run for quite a long time and likely fail. Usually you don't actually need `<windows.h>`, however. Check out the `ignored_headers` and `ignore_system_headers` parameters of `build_lib()` for two ways to ignore headers.

When there's an error while parsing part of a header, NiceLib should spit out the "chunk" of code that caused a parse error. Usually this is due to some compiler-specific syntax that `cffi` does not understand and you simply want to remove, like a `__declspec`.

To remove the problematic syntax, you should use NiceLib's parser hooks, which are used to transform a stream of C tokens (token hooks) or a C abstract-syntax-tree (AST hooks). These hooks get passed into `build_lib()` via its `token_hooks` and `ast_hooks` arguments. There are a few hooks which are enabled by default since they're required so often.

NiceLib already has built-in hooks for many common cases, so be sure to check out *Token Hooks* and *AST Hooks* before writing your own. Take a look at what hooks are available, it will give you a sense of what types of fixes are usually required.

If you do have an unusual case and need to write your own hook, be sure to check out the *Token Hook Helpers* and *AST Hook Helpers*, which can be used to simplify the process.

4.1.3 Behind the Scenes

`build_lib()` does a few things when it's executed. First, it looks for the header(s) in the locations you've specified and invokes `process_headers()`, which preprocesses the headers and returns two strings: the cleaned header C code and the extracted macros, converted to Python code. It uses the cleaned header to generate an out-of-line `cffi` module, then appends code for loading the shared lib and implementing the headers' macros. This finished module can be imported like any other, but is usually loaded via `load_lib()`.

How Headers are Processed

The bulk of the heavy lifting is done (and most issues are most likely to occur) in `process_headers()`. First, the header code is tokenized and parsed by a lexer and parser defined in the `process` module. This parser doesn't understand C, but does understand the language of the C preprocessor. It keeps track of macro definitions, removing them from the token stream and performing expansion of macros when they are used. It also understands and obeys other directives, including conditionals and `#includes`. After parsing, the token stream should be free of any harmful directives that `pycparser`/`cffi` don't understand.

This token stream can then be acted upon by the so-called "token hooks", which can be supplied via the arguments lists of `process_headers()` and `build_lib()`. These hooks are functions which both accept and return a sequence of tokens. The purpose of each hook is to perform a specific transformation on the token stream, usually removing nonstandard syntax that `pycparser`/`cffi` may not understand (e.g. C++ specific syntax).

Once the hooks are all applied, the tokens are joined together into chunks that are parseable by `pycparser`'s C parser. After each chunk is parsed, it is acted upon by the "AST hooks", which take the parsed abstract syntax tree (AST) and a reference to the parser and return a transformed AST. This allows hooks to modify the AST and the state of the parser. Once all of the chunks have been parsed and joined together into one big AST, this tree is used to generate the C source code which is later returned by `process_headers()`.

4.2 Creating Mid-Level Bindings

`NiceLib` is the base class that provides a nice interface for quickly defining mid-level library bindings. You define a subclass for each specific library (.dll/.so file) you wish to wrap. `NiceLib`'s metaclass then converts your specification into a wrapped library. You use this subclass directly, without instantiating it.

4.2.1 What Are Mid-Level Bindings?

It's worth discussing what we mean by "mid-level" bindings. Mid-level bindings have a one-to-one correspondence between low-level functions and mid-level functions. The difference is that each mid-level function has a more Pythonic interface that lets the user mostly or entirely avoid working with `cffi` directly. In other words, the overall structure of the library stays the same, but each individual function's interface may change.

These mid-level bindings can then be used to craft high-level bindings that might have a completely different structure than the underlying low-level library.

Let's say we want to wrap a motor-control library and its header looks something like this:

```
// Example header file
typedef void* HANDLE;

int GeneralGetDeviceList(uint* devList, uint listSize);
void GeneralGetErrorString(int errCode, char *recvBuf, uint bufSize);
int GeneralOpenMotor(uint motorID, HANDLE *phMotor);

int MotorClose(HANDLE hMotor);
int MotorMoveTo(HANDLE hMotor, long pos);
int MotorGetPosition(HANDLE hMotor, long *pPos);
int MotorGetSerial(HANDLE hMotor, char *recvBuf, uint bufSize);
```

We would then write bindings like this:

```
from nicelib import load_lib, NiceLib, Sig, NiceObject, RetHandler, ret_ignore

@RetHandler(num_retvals=0)
def ret_errcode(retval):
    if retval != 0:
        raise MotorError(NiceMotor.GetErrorString(retval))

class NiceMotor(NiceLib):
    _info_ = load_lib('awesomemotor', __package__)
    _ret_ = ret_errcode
    _prefix_ = 'General'

    GetDeviceList = Sig('arr', 'len=20')
    GetErrorString = Sig('in', 'buf', 'len', ret=ret_ignore)
    OpenMotor = Sig('in', 'out')

    class Motor(NiceObject):
        _init_ = 'OpenMotor'
        _prefix_ = 'Motor'

        Close = Sig('in')
        MoveTo = Sig('in', 'in')
        GetPosition = Sig('in', 'out')
        GetSerial = Sig('in', 'buf', 'len=64')
```

Then we can use the library like this:

```
motor_ids = NiceMotor.GetDeviceList()

for motor_id in motor_ids:
    motor = NiceMotor.Motor(motor_id)
    pos = motor.GetPosition()
    serial = motor.GetSerial()
    print("Motor {} is at position {}".format(serial, pos))
    motor.Close()
```

There are a number of features in use in this example: prefix removal, return value wrapping, array and string buffer output, and `NiceObjectDefs` with custom initializers. These make use of settings, which you can read more about below.

4.2.2 Settings

Settings, also called flags, give you extra control over how a library is wrapped. Settings are scoped, meaning that you can specify them on a class-wide, NiceObject-wide, and per-function basis. For example:

1. **Class-level:** Give the NiceLib class an attribute with the setting name surrounded by single underscores:

```
class MyLib(NiceLib):
    _buflen_ = 128
```

2. **NiceObject-level:** Give the NiceObject class an attribute with the setting name surrounded by single underscores:

```
class MyLib(NiceLib):
    class MyObject(NiceObject):
        _buflen_ = 128
```

3. **Function-level:** Pass settings as keyword args to the `Sig` constructor:

```
MyFunction = Sig('in', 'in', 'out', buflen=128)
```

The available settings are:

prefix A `str` or sequence of `str`s specifying prefixes to strip from the library function names. For example, if the library has functions named like `SDK_Func()`, you can set `_prefix_` to `'SDK_'`, and access them as `Func()`. If multiple prefixes are given, they are tried in order for each signature until the appropraite function is found. The empty prefix `''` is always tried. Sometimes you may want to specify one library-wide prefix and a different per-object prefix, as done in the above example.

These prefixes also get stripped from macro names and enum constants.

ret A function or `str` specifying a handler function to handle the return values of each library function. See *Return Value Handlers* for details.

buflen An `int` specifying the default length for buffers and arrays. This can be overridden on a per-argument basis in the argument's spec string, e.g. `'len=64'` will make a 64-character buffer or a 64-element array.

free_buf A function that is called on the pointer returned for 'bufout' argtypes, used for freeing their associated memory. It is called immediately after the buffer is copied to produce a Python string, but is not called if a null pointer is returned. May be None.

use_numpy If True, convert output args marked as `'arr'` to numpy arrays. Requires numpy to be installed.

struct_maker A function that is called to create an FFI struct of the given type. Mainly useful for odd libraries that require you to always fill out some field of the struct, like its size in bytes.

4.2.3 `NiceLib` Class Attributes

`NiceLib` subclasses make use of a few underscore-surrounded special class attributes. In addition to the class-wide *settings* described above, they include:

info A `LibInfo` object that contains access to the underlying library and macros. Required (unless you are using the old-style `_ffi_`, `_ffilib_`, and `_defs_` attributes)

Typically you will want to pass the relevant library attributes via a `LibInfo` instance created using `load_lib()`, as shown in the examples above. However, it is currently possible to specify them directly. This was the original method, but may become deprecated in later versions of `NiceLib`.

ffi FFI instance variable. Required if not using `_info_`.

ffilib FFI library opened with `ffi.dlopen()`. Required if not using `_info_`.

defs `dict` containing the Python-equivalent macros defined in the header file(s). Optional and only used if not using `_info_`.

4.2.4 Function Signatures

Function signatures are specified as `Sig` class attributes. A `Sig`s positional args are strings that define the input-output signature of the underlying C function. Per-function settings, like custom return value handling, are passed as keyword args.

It's important to note that a `Sig` is designed to closely match the signature of its C function, i.e. there's a one-to-one correspondence between arg strings and C function args.

The basic idea behind signature specifications is to handle input and output in a more Pythonic manner—inputs get passed in via a function's arguments, while its outputs get returned as part of the function's return values. Take the simple example from above:

```
OpenMotor = Sig('in', 'out')
```

This says that the C function's first argument (`uint motorID`) is used strictly as input, and its second argument (`HANDLE *phMotor`) is used strictly as output—the function takes an ID number and returns a handle to a newly opened motor. Using this signature allows us to call the function more naturally as `handle = OpenMotor(motorID)`.

The available signature values are:

'in' The argument is an input and gets passed into the mid-level function.

'out' The argument is an output. It is not passed into the mid-level function, but is instead added to the list of return values. NiceLib automatically allocates an appropriate data structure, passes its address-pointer to the C function, uses the dereferenced result as the return value.

This can't be used for `void` pointers, since there's no way to know what to allocate, or what type to return.

'inout' The argument is used as both input and output. The mid-level function takes it as an argument and also returns it with the return values. You can pass in either a value or a pointer to the value. For example, if the underlying C argument is an `int *`, you can pass in a `cffi int` pointer, which will be used directly, or (more typically) you can pass in a Python int, which will be used as the initial value of a newly-created `cffi` int pointer.

'bufout' The argument is a pointer to a string buffer (a `char**`). This is used for when the C library creates a string buffer and returns it to the user. NiceLib will automatically convert the output to a Python `bytes`, or None if a null pointer was returned.

If the memory should be cleaned up by the user (as is usually the case), you may use the `free_buf` setting to specify the cleanup function.

'buf' The argument is a string buffer used for output. The C argument is a `char` pointer or array, into which the C-function writes a null-terminated string. This string is decoded using `ffi.string()`, and added to the return values.

This is used for the common case of a C function which takes both a string buffer and its length as inputs, so that it doesn't overrun the buffer. As such, `'buf'` requires a corresponding `'len'` entry. The first `'buf'`/`'arr'` is matched with the first `'len'` and so forth. If you don't need to pass in a length parameter to the C-function, use `'buf[n]'` as described below.

NiceLib will automatically create the buffer and pass it and the length parameter to the C-function. You simply receive the `bytes`.

'buf[n]' The same as `'buf'`, but does not have a matching `'len'`. Because of this, the buffer length is specified directly as an int. For example, a 20-char buffer would be `'buf[20]'`.

'arr' The same as `'buf'`, but does not call `ffi.string()` on the returned value. Used e.g. for `int` arrays.

'arr[n]' The same as `'buf[n]'`, but does not call `ffi.string()` on the returned value. Used e.g. for `int` arrays.

'len' The length of the buffer being passed to the C-function. See `'buf'` for more info. This will use the length given by the innermost `buflen` setting.

'len=n' The same as `'len'`, but with an overridden length. For example, `'len=32'` would allocate a buffer or array of length 32, regardless of what `buflen` is.

'len=in' Similar to `'len=n'`, except the mid-level function takes an input argument which is an `int` specifying the size of buffer that should be allocated for that invocation.

'ignore' Ignore the argument, passing in 0 or NULL, depending on the arg type. This is useful for functions with "reserved" arguments which don't do anything.

4.2.5 Return Value Handlers

`RetHandlers`, which specify functions to handle the return values of each library function, are given via the `ret` flag, as mentioned in *Settings*. Return handlers are created by using the `@RetHandler` decorator—for example, the built-in `ret_return` handler is defined thusly:

```
@RetHandler(num_retvals=1)
def ret_return(retval):
    return retval
```

`num_retvals` indicates the number of values that the handler returns, which is often zero. Return handlers can be used to raise exceptions, return values, or even do custom handling based on what args were passed to the function.

A handler function takes the C function's return value—often an error/success code—as its first argument (see below for other optional parameters it may take). If the handler returns a non-None value, it will be appended to the wrapped function's return values.

Builtin Handlers

There are two handlers that nicelib defines for convenience:

`ret_return()` The default handler. Simply appends the return value to the wrapped function's return values.

`ret_ignore()` Ignores the value entirely and does not return it. Useful for `void` functions

Injected Parameters

Sometimes it may be useful to give a handler more information about the function that was called, like the C parameters it was passed. If you define your handler to take one or more specially-named args, they will be automatically injected for you. These include:

funcargs The list of all `cffi`-level args (including output args) that were passed to the C function

niceobj The `NiceObject` instance whose method was called, or None for a top-level function

4.2.6 NiceObjects

Often a C library exposes a distinctly object-like interface like the one in our example. Essentially, you have a handle or ID of some resource (a motor in the example), which gets passed as the first argument to a subset of the library's functions. It makes sense to treat these functions as the *methods* of some type of object. NiceLib allows you to define these types of objects by subclassing `NiceObject`.

`NiceObject` class definitions are nested inside your `NiceLib` class definition, and consist of method `Sig`s and object-specific settings. When you instantiate a `NiceObject`, the args are passed to the `NiceObject`'s *initializer*, which returns a handle. This handle is passed as the first parameter to all of the `NiceObject`'s "methods". This initializer is specified using the `NiceObject`'s `_init_` class attribute, which can be either a function or the name of one of the mid-level functions (as with `'OpenMotor'` in the example above). If `_init_` is not defined, the args passed to the `NiceObject`'s constructor are used directly as the handle.

Without using `_init_`, object construction would look like this:

```
handle = MyNiceLib.GetHandle()
my_obj = MyNiceLib.MyObject(handle)
my_obj.AwesomeMethod()
```

But if we use `_init_`:

```
class MyNiceLib(NiceLib):
    [...]
    GetHandle = Sig('out')

    class MyObject(NiceObject):
        _init_ = 'GetHandle'
        [...]
```

we can then do this:

```
my_obj = MyNiceLib.MyObject()
my_obj.AwesomeMethod()
```

and bypass passing around handles at all.

Multi-value handles

Usually an object will have only a single value as its handle, like an ID. In the unusual case that you have functions which take more than one value which act as a collective 'handle', you should specify this number as `_n_handles_` in your `NiceObject` subclass.

4.2.7 Auto-Generating Bindings

If nicelib is able to parse your library's headers successfully, you can generate a convenient binding skeleton using `generate_bindings()`.

4.3 API Documentation

These are the major NiceLib classes and functions of which you should know:

- *Header Processing API*

4.3.1 Header Processing API

Token Hooks

These functions can all be used in the `token_hooks` passed to `build_lib()` or `process_headers()`

Token Hook Helpers

These functions and classes are useful for writing your own custom token hooks:

AST Hooks

AST Hook Helpers

4.3.2 Mid-Level Binding API