# A Python Wrapper Code Generator for Dynamic Libraries

**Arne Bachmann**
*German Aerospace Center (DLR), Cologne, Germany*
arne.bachmann@dlr.de

**Abstract**
We introduce a new Python code generator for conveniently and transparently wrapping native dynamic libraries. The presented code generator is used in several projects for scientific collaboration and can be adapted to other projects fairly easily.

## 1. Introduction

In research institutions, many scientists and engineers of manifold disciplines collaborate to build optimized models.

For efficiency and ease of use, scientists nowadays more and more become aware of the need to unify their software environments. They use software integration frameworks and common libraries to minimize the potential of observing incompatibilities and interfaces mismatch. The introduction of these tools for collaboration guided by experts in software technology is recommended.

On the other hand, each scientific discipline has its own specialized tools and methodologies needed to calculate models and find optimized solutions in their field of excellence. A good solution for allowing flexibility, while keeping software diversification at a necessary minimum, is the introduction of a common and easy-to-learn scripting language.

Python is one of those candidate languages and has significantly gained popularity among scientists over the past years [1] and offers both a very clear and non-verbose syntax, and a host of useful domain-specific modules to support almost every task its users may have.

While scripting abilities are useful and convenient for researchers, they also often write their own highly runtime-optimized and parallelized software in compiled languages like Fortran and C/C++. These codes can be linked statically into executable files and deployed as stand-alone tools available through networks to the research community, but can also be packaged into dynamic libraries that can be linked during runtime by some other code, which may use whatever functions it needs from the library. Under Windows operating systems, these are dynamic-link library files `*.dll` while under Linux there are shared object files `*.so` [2].

The combination we observe in our interdisciplinary projects at DLR is that there are a few widely used, but specialized and optimized libraries that are shared between institutes when collaboratively developing optimized models. The usage of functions from these libraries varies widely between users and use cases, therefore a lot of researchers use those libraries from within Python, where they have freedom of implementation, combined with the power of precompiled optimized functions.

## 2. The wrapper code generator explained

One of the things we did at the department of Simulation and Software Technology at DLR was to create a library wrapper generator for dynamic libraries to be used in the Python language.

The advantage of this approach is obvious: By writing the wrapper generator once, we gained the benefit of wrapping not only one library, but all of our libraries at once, and it can be used by third parties, too, due to its open source license. The second advantage is automation: Each time we extend the core libraries and introduce new functions, the wrapper generator can automatically update the wrapper code, too, disregarding spurious manual annotations that might become necessary for exotic cases. This boils down to almost a free lunch leading to wrapper codes to be in sync with the core libraries at all times.

We investigated other approaches like the SWIG [3] or the suggested procedure presented in [4], which is very similar to the approach presented here. The first allows for the same bindings definition to be used in several programming languages, but needs a manual compilation step for each language supported, while the second needs several binary tools to parse the header file and extract the interface definition of the library to wrap into an XML representation. In our projects we took a Python-only approach, based on a pure-Python parsing and generator of `ctypes`-wrapping code.

For the time being, the wrapper generator supports the following features:

- A usable enum representation in Python code,
- Pure-Python values for anything going in and coming out of C-type functions,
- Multidimensional array handling in pure python for all basic data types,
- Automatic recognition of input and output parameters, wherever possible,
- Signature augmentation of necessary array dimension parameters, where needed,
- Generation of Windows installers and Linux archives by means of `distutils`,
- Upload facilities to repository sites by calling external modules like `googlecode_upload`.

The wrapper generator called `make-wrapper.py` can be broken down into the following parts:

- A wrapper code template (`wrapperstub.py`),
- A header file parser,
- The code generator,
- Management code.

The code template is the core of the generated wrapper code and contains state management and exception handling. It also encapsulates all library loading with the `ctypes` module.

The header file parser reads in the header file (*.h) associated with the library to wrap. By applying simple pattern matching mechanisms in functions like `findLinesWith`, all relevant parts of the file are extracted and parsed. This results in an object-oriented representation of the function signatures like `ParamType` and `Param`, including information on pointers, data types and argument names. When adapting the wrapper generator to your own libraries, you might need to adapt the parsing patterns to your source code conventions. Functions like `createEnums` and `createMethodStubs` are the heart of the wrapper generator. The following code fragments show some of the simple mechanisms in the wrapper generator:

```
def getPartBefore(contents, term):
    ''' Iterator to read until a certain line. doctest code was omitted here. '''
    for line in contents:
        if term in line:
            return
        yield line

def isPragma(line):
    ''' Check if this is a pragma line to ignore. doctest code omitted here. '''
    return re.match("\\A\\s*#", line) != None
```

The code generator itself is the core mechanism that generates valid Python code to be merged with the template. Enums are read in and written out in a Pythonesque way, file and configuration handles are recognized and removed from the signatures, and all functions (or methods) of the library are wrapped by converter codes that handle all mapping to and from C-type value representations, including array creation.

As an example, we show the original C interface and the generated code thereof:

```
/**
  @brief Retrieves the names of all dimensions.
  #PY:1:-1#  -1 means one user specified return array (of strings)
 */
DLL_EXPORT ReturnCode tixiGetArrayDimensionNames(const TixiDocumentHandle handle,
                                                 const char *arrayPath, char **dimensionNames);


def getArrayDimensionNames(self, arrayPath, _num_0_):
    ''' This is the generated Python version of the above C header.
        This comment was manually added. '''
    _c_arrayPath = c_char_p()                      # ctypes char*
    _c_arrayPath.value = arrayPath                 # assign the value
    _d_dimensionNames = c_char_p * (1 * _num_0_)   # create a datatype for the output array
    _c_dimensionNames = _d_dimensionNames()        # create the variable to hold the output values
    tixiReturn = self.TIXI.tixiGetArrayDimensionNames(self._handle, _c_arrayPath,
                                                      byref(_c_dimensionNames))
    self._validateReturnValue(tixiReturn, arrayPath, _num_0_)  # check the return code
    return tuple([__x__ for __x__ in _c_dimensionNames])       # return pure python values
```

The management code comprises only of a few calls to the `distutils` module for installer creation and an upload script to the popular source code repository found at `googlecode.com`. This way by invoking the wrapper generator with a command line argument like `--upload` not only the wrapper gets generated, but also its final installers are uploaded instantly to their web sites. A typical wrapper execution looks like that:

```
N:\svn\tixi\Src> python make-wrapper.py
Found 35 values for enum ReturnCode
Found 2 values for enum StorageMode
Found 2 values for enum OpenMode

Building source distribution
running sdist
...
Building windows binary distribution
Running bdist_wininst
...

N:\svn\tixi\Src>
```

## 3. Technical details

The generated code depends only on the `sys` and `ctypes` modules. The template code contains string formatting statements like `%s` that are replaced by the generated codes in a straight-forward manner. The current code base is for the Python 2 family, adaption to Python 3 is a major task that leads to incompatible code changes in both directions which would constitute a completely new project. Since there is no real user base for the Python 3-family at the time being, we neglect upwards compatibility at the moment.

Enums are represented as class attributes with a string value equal to the enum entry's label. This makes using them very literal in the user’s source code and allows for string serialization at the same time. For deserialization there is also a backwards-mapped dictionary called `_names` within the enum class automatically generated.

Parsing is done mostly line-wise; the whole header file is read into memory for faster processing. The template code contains functions for finding certain character sequences within the header file in functions like `getPartBefore` and `getPartAfter`. These simple and not very runtime-efficient helper functions are fully sufficient to find and separate all relevant parts of the header file, unless a uniform code formatting has been neglected, which would be a bad code smell anyway.

The code generator parses the found method signatures and analyses them according to their constitution: Parameters that contain at least one asterisk (*) are considered an array, unless they are output parameters where the C-type library returns newly created values. One additional but uncritical challenge are strings and string arrays, because in C there is no way to distinguish between a pointer to a character array `char*` and a string. The same applies to `char**`, which can be a two-dimensional character array, but also an array of strings. In fact the rules for determination of arrays and pointers are even more difficult than shown here and details can be found in the source code.

For an unbiased interpretation of pointers, arrays and data types, a convention and a short markup were introduced. The convention goes like this:

- The first parameter – if it is some kind of DLL handle – is ignored and doesn’t count against the parameters. It is handled inside the code template and the generator code,

- The last parameter is by default the only output parameter, unless it doesn't have an asterisk or unless noted otherwise in the annotation,
- All other parameters are considered input parameters, either of singular or array type.

The annotation for function signatures is parsed from the comment block residing above each function declaration, from the same place where other documentation stems from, as for example, markup for the Doxygen processor and the description of Fortran bindings. The syntax for the wrapper annotation is as follows:

START DELIMITER OUTPUTS DELIMITER ARRAYS END

Where START is the character sequence "#Py", DELIMITER is ":", and END is "#".

OUTPUTS is a list of comma-separated integer numbers in the range of [0; number of parameters).

ARRAY is a list of comma-separated lists of semicolon-separated integer numbers in the range of [0; number of parameters).

The *outputs* list enumerates all parameter positions where an output from the library is expected. This list is only necessary to note in the header file if there is more than one output pointer or the only parameter is not at the last (right-most) position. As an example may serve a geometric function that converts one 32-bit integer value into three double values representing one combined Cartesian coordinate. Here we might find an annotation like "#PY:1,2,3#" which tells us that the (last) three parameters at positions 1, 2 and 3 are output parameters, while the first is not (at position 0).

The arrays list became necessary for functions that not only write data to pointers that are put into the function (created by the `ctypes` module), but for functions that allocate new memory portions and return the created pointer(s) to the wrapper code. Here the wrapper generator needs to know the size of the array generated within the library in advance, because the `ctypes` module cannot expand or shrink arrays dynamically. The semantics of the semicolon-separated lists are as follows:

- Each list represents one of the outputs, from left to right mapped to the parameters in the outputs list,
- A value of -1 in the arrays list means user-provided: The wrapper generator will add one additional parameter to the Python wrapper code, which the user must fill in to tell the wrapper about the expected size of the returned array,
- Any number of positive integer values (including zero) in the inner list is considered as indexes of input parameter positions. The product of all their input values is calculated and taken as output array size. This is useful in cases where, e.g., a two-dimensional matrix is created or calculated and the wrapper code needs the overall size of the generated array.

With these simple conventions and one small annotation in diverging cases, we gained enough flexibility for all our use cases as of today, while keeping a lean code base regarding Python code.

Drawbacks of the simplicity of this approach are for example the inability to automatically and recursively go through all included header files from the wrapped file, no knowledge about `#define` pragmas or macros.

Besides the other advantages, we don't need "monkey-patching" as in the approach taken by Kloss, there are no complex command-line parameters and no external dependencies other than the adapted stub.

## 4. Summary and outlook

We have presented a simple yet very useful Python code generator that almost fully automatically wraps dynamic Fortran or C/C++ libraries via the `ctypes` module, while completely shielding users from all intricacies of C-types handling.

Automation of wrapper code generation is maximized by one basic convention, and flexibility is reinstated for diverging cases by introduction of a short annotation within the documentation block of a function declaration inside the header file.

The wrapper generator has been in use successfully at DLR by several projects and institutes of diverse research areas collaborating in airplane modeling and optimization. Only minimal adjustments are necessary for adaption to libraries in other projects or different coding conventions and we hope to see other projects making use of the library in the future.

Ideas for further improvement include automatic generation of `doctest` tests in the generated code to reliably ensure code correctness. Currently there are embedded tests only inside the generator code.

## References

[1] *TPCI History for language Python*. Available at http://www.tiobe.com/index.php/paperinfo/tpci/Python.html

[2] *Static, Shared Dynamic and Loadable Linux Libraries*. Available at http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

[3] *Simplified Wrapper and Interface Generator (SWIG)*. Available at http://www.swig.org

[4] Kloss, Guy K. (2007). *Automatic C Library Wrapping - Ctypes from the Trenches.* The Python Papers 3(3): 5.

Libraries that currently use the Python DLL wrapper code generator:

- http://tixi.googlecode.com
- http://tigl.googlecode.com