zugbruecke Documentation

Release 0.0.15

Sebastian M. Ernst

Jul 10, 2020

Contents

zugbrücke

Calling routines in Windows DLLs from Python scripts running under Linux, MacOS or BSD.

zugbruecke is a drop-in replacement for *ctypes* with minimal, *ctypes*-compatible syntax extensions. This manual describes what makes *zugbruecke* special and how it differs from *ctypes*. It does NOT substitute the ctypes documentation. Please read the latter first if you have never called foreign functions with *ctypes* from *Python* scripts.

CHAPTER 1

Introduction

1.1 Synopsis

zugbruecke is an **EXPERIMENTAL** *Python* module (currently in development *status 3/alpha*). It allows to call routines in *Windows* DLLs from *Python* code running on Unices / *Unix*-like systems such as *Linux*, *MacOS* or *BSD*. *zugbruecke* is designed as a drop-in replacement for *Python*'s standard library's *ctypes* module. *zugbruecke* is built on top of *Wine*. A stand-alone *Windows Python* interpreter launched in the background is used to execute the called DLL routines. Communication between the *Unix*-side and the *Windows*/*Wine*-side is based on *Python*'s build-in multiprocessing connection capability. *zugbruecke* has (limited) support for pointers, struct types and call-back functions. *zugbruecke* comes with extensive logging features allowing to debug problems associated with both itself and with *Wine*. *zugbruecke* is written using *Python* 3 syntax and primarily targets the *CPython* implementation of *Python*.

1.2 Motivation

Academic interest and frustration over the lack of a project of this kind, mostly. *zugbruecke* ultimately started as an attempt to collect and consolidate a sizable collection of "ugly hacks" accumulated over the years. Those had mostly been written for accessing routines for complicated numerical computations in proprietary DLLs on Linux clusters.

The need for calling individual routines offered by DLLs from *Linux*/*MacOS*/*BSD* software/scripts is reflected in numerous threads in forums and mailing lists reaching back well over a decade. The recommended approach so far has been (and still is!) to write a *Wine* application, which links against `winelib`, thus allowing to access DLLs. Wine applications can also access libraries on the *Unix* "host" system, which provides the desired bridge between both worlds. Nevertheless, this approach is anything but trivial. *zugbruecke* is supposed to satisfy the desire for a "quick and dirty" solution for calling routines from a high level scripting language, *Python*, directly running on the *Unix* "host" system. With respect to "quick", *zugbruecke* works just out of the box with *Wine* installed. No headers, compilers, cross-compilers or any other configuration is required - one import statement followed by well established `ctypes` syntax is enough. It is pure *Python* doing its job. With respect to "dirty", well, read the project's documentation from start to finish.

1.3 Implementation

During the first import of *zugbruecke*, a stand-alone *Windows*-version of the *CPython* interpreter corresponding to the used *Unix*-version is automatically downloaded and placed into the module's configuration folder (by default located at `~/.zugbruecke/`). Next to it, also during first import, *zugbruecke* generates its own *Wine*-profile directory for being used with a dedicated `WINEPREFIX`. This way, any undesirable interferences with other *Wine*-profile directories containing user settings and unrelated software are avoided.

During every import, *zugbruecke* starts the *Windows Python* interpreter on top of *Wine*. It is used to run a server script named `_server_.py`, located in the module's folder.

zugbruecke offers everything *ctypes* would on the *Unix* system it is running on plus everything *ctypes* would offer if it was imported under *Windows*. Functions and classes, which have a platform-specific behavior, are replaced with dispatchers. The dispatchers decide whether the *Unix* or the *Windows* behavior should be used depending on the context of how they were invoked and what parameters were passed into them. If *Windows* specific behavior is chosen, calls are passed from *zugbruecke*'s client code running on *Unix* to the server component of *zugbruecke* running on *Wine*.

1.4 Use cases

- Quickly calling routines in proprietary DLLs.
- Reading legacy file formats and running mission critical legacy plugins for legacy ERP software in a modern environment comes to mind.
- Calling routines in DLLs which come, for some odd reason like "developer suddenly disappeared with source code", without source code. DLLs found in company-internal software or R&D projects come to mind.
- More common than one might think, calling routines in DLLs, of which the source code is available but can not be (re-)compiled (on another platform) / understood / ported for similarly odd reasons like "developer retired and nobody knows how to do this" or "developer 'went on' and nobody manages to understand the undocumented code". The latter is especially prevalent in academic environments, where what is left of years of hard work might only be a single "binary blob" - a copy of an old DLL file. All sorts of complicated and highly specialized numerical computations come to mind.

CHAPTER 2

Installation

2.1 Getting *Wine*

For using *zugbruecke*, you need to install **Wine** first. Depending on your platform, there are different ways of doing that.

- Installation instructions for various Linux distributions
- Installation instructions for Mac OS X
- Installation instructions for FreeBSD

2.2 Getting *zugbruecke*

The latest (more or less) **stable release version** can be installed with *pip*:

```
pip install zugbruecke
```

If you are interested in testing the latest work from the **development branch**, you can try it like this:

```
pip install git+https://github.com/pleiszenburg/zugbruecke.git@develop
```

2.3 Installing *zugbruecke* in development mode

If you are interested in contributing to *zugbruecke*, you might want to install it in development mode. You can find the latest instructions on how to do this in the CONTRIBUTING file of this project on *Github*.

CHAPTER 3

Examples

A lot of code, which was written with `ctypes`' `cdll`, `windll` or `oledll` in mind and which runs under *Windows*, should run just fine with *zugbruecke* on Unix.

```
import zugbruecke as ctypes

simple_demo_routine = ctypes.windll.LoadLibrary('demo_dll.dll').simple_demo_routine
simple_demo_routine.argtypes = (ctypes.c_float, ctypes.c_float)
simple_demo_routine.restype = ctypes.c_float
return_value = simple_demo_routine(20.0, 1.07)
print('Got "%f".' % return_value)
```

It will print `Got "1.308412".` assuming that the corresponding routine in the DLL looks somewhat like this:

```
float __stdcall __declspec(dllimport) simple_demo_routine(float param_a, float param_
↪b)
{ return param_a - (param_a / param_b); }
```

Because of the drop-in replacement design of *zugbruecke*, it is possible to write *Python* code which works under both *Unices* and *Windows*.

```
from sys import platform
if any([platform.startswith(os_name) for os_name in ['linux', 'darwin', 'freebsd']]):
        import zugbruecke as ctypes
elif platform.startswith('win'):
        import ctypes
else:
        # Handle unsupported platforms
```

For more examples check the examples directory of this project. For an overview over its entire range of capabilities have a look at zugbruecke's test suite. For the full demo DLL source code check the demo_dll directory of this project.

Please also consult the documentation of ctypes.

CHAPTER 4

The session model

zugbruecke operates based on a session model. Every session represents a separate *Windows Python* interpreter process running on *Wine*. By default, *zugbruecke* starts one session during import, but the user can start more sessions if required. Sessions are identified by a unique (hash) ID string. Sessions have a life-cycle and require termination routines to run before they can be dropped or deleted. By default, sessions terminate themselves automatically when the *Python* interpreter quits.

4.1 Class: `zugbruecke.session`

By creating an instance of this class, a new session can be started. The number of instances/sessions is only (theoretically) limited by the amount of available memory and by the number of available network ports on the host system (two ports per instance are required). The *constructor can be configured*.

4.1.1 Method: `load_library`

Parameters:

- `dll_name` (str)
- `dll_type` (str)
- `dll_param` (dict, optional)

Return value:

- An instance of *zugbruecke*'s internal DLL representation, which mimics instances of *ctypes*' `CDLL`, `WinDLL` or `OleDLL` classes depending on context.

The second parameter, `dll_type`, determines the calling convention and can be set to "cdll", "windll" or "oledll". Any other value will raise an error.

The first parameter, `dll_name`, will be passed directly into the *ctypes* library class constructor on the *Wine* side, i.e. `CDLL`, `WinDLL` or `OleDLL`. All limitations and features of those constructors apply, e.g. the possibility of referring to DLLs without their .dll-file-extension or the use of absolute or relative *Windows* paths. *zugbruecke* offers *methods*

for path conversion if required. If *ctypes* on the *Wine* side raises an error, e.g. because the DLL can not be found, the error will be re-raised by *zugbruecke* on the Unix side.

The third parameter is optional and allows to pass a dict with the following keys:

- `mode`
- `use_errno`
- `use_last_error`

Those can be used to pass values into the corresponding parameters of the ctypes constructors.

4.1.2 Method: `set_parameter`

Parameters:

- `parameter` (dict)

Used to *re-configure* a running session. Accepts a dictionary containing *configuration parameters*.

4.1.3 Method: `terminate`

This method can be used to manually terminate a session. It will quit the *Windows Python* interpreter running in the background. Once terminated, a session can not be re-started. Any handles on DLLs and their routines derived from this session will become useless.

4.1.4 String: `id`

Allows to read the unique session id.

4.1.5 Boolean: `up`

Can be read to determine whether a session is up. Once a session is terminated, it will be set to `False`.

4.2 Instance: `zugbruecke.current_session`

This is the default session of *zugbruecke*. It will be started during import. Like every session, it can be *re-configured* during run-time. If any of the usual *ctypes* members are imported from *zugbruecke*, like for instance `cdll`, `CDLL`, `CFUNCTYPE`, `windll`, `WinDLL`, `WINFUNCTYPE`, `oledll`, `OleDLL`, `FormatError`, `get_last_error`, `GetLastError`, `set_last_error` or `WinError`, this session will be used.

CHAPTER 5

Handling pointers

5.1 The memory synchronization protocol

Because *zugbruecke* runs code in a separate *Python* interpreter on *Wine*, pointers pose a special kind of problem. They can, unfortunately, not be passed from the code running on the *Unix* side to the code running on the *Wine* side or vice versa. This is why the memory (to where pointers are pointing) must be kept in sync on both sides. The memory synchronization can be controlled by the user through the `memsync` protocol. `memsync` implements special directives, which do not interfere with *ctypes* should the code be required to run on *Windows* as well.

zugbruecke can handle some types of pointers on its own, without additional `memsync` directives. Pointers to variables containing a single element (e.g. a floating pointer number or a structure) and pointers to fixed-length arrays are handled transparently without additional directives. If, on the other hand, the size of the memory a pointer is pointing to is dynamically determined at runtime, *zugbruecke* must be provided with a hint on where it can find information on the size of the memory section within the arguments or return value of a routine call. Those hints can be provided through `memsync`.

5.2 A simple example: An array of floating point numbers of variable length

Consider the following example DLL routine in C:

```
void __stdcall __declspec(dllimport) bubblesort(
        float *a,
        int n
        )
{
        int i, j;
        for (i = 0; i < n - 1; ++i)
        {
                for (j = 0; j < n - i - 1; ++j)
                {
```

(continues on next page)

(continued from previous page)

```
                        if (a[j] > a[j + 1])
                        {
                                float tmp = a[j];
                                a[j] = a[j + 1];
                                a[j + 1] = tmp;
                        }
                }
        }
}
```

It is a simple implementation of the "bubblesort" algorithm, which accepts a pointer to an array of floats of arbitrary length and an integer with length information. The array is being sorted within its own memory, so the caller expects a sorted array at the passed pointer after the call.

With *ctypes* on *Windows*, you could call it like this:

```
from ctypes import windll, cast, pointer, POINTER, c_float, c_int

__dll__ = windll.LoadLibrary('demo_dll.dll')
__bubblesort__ = __dll__.bubblesort
__bubblesort__.argtypes = (POINTER(c_float), c_int)

def bubblesort(values):

        ctypes_float_values = ((c_float)*len(values))(*values)
        ctypes_float_pointer_firstelement = cast(
                pointer(ctypes_float_values), POINTER(c_float)
                )
        __bubblesort__(ctypes_float_pointer_firstelement, len(values))
        values[:] = ctypes_float_values[:]

test_vector = [5.74, 3.72, 6.28, 8.6, 9.34, 6.47, 2.05, 9.09, 4.39, 4.75]
bubblesort(test_vector)
```

For running the same code with *zugbruecke* on *Unix*, you need to add information on the memory segment representing the array. This is done by adding another attribute, `memsync`, to the `__bubblesort__` function handle (just like you usually specify `argtypes` and/or `restype`). The following example demonstrates how you must modify the above example so it works with *zugbruecke*:

```
from zugbruecke import windll, cast, pointer, POINTER, c_float, c_int

__dll__ = windll.LoadLibrary('demo_dll.dll')
__bubblesort__ = __dll__.bubblesort
__bubblesort__.argtypes = (POINTER(c_float), c_int)
__bubblesort__.memsync = [
        {
                'p': [0],
                'l': [1],
                't': 'c_float'
                }
        ]

def bubblesort(values):

        ctypes_float_values = ((c_float)*len(values))(*values)
        ctypes_float_pointer_firstelement = cast(
```

(continues on next page)

(continued from previous page)

```
                pointer(ctypes_float_values), POINTER(c_float)
                )
        __bubblesort__(ctypes_float_pointer_firstelement, len(values))
        values[:] = ctypes_float_values[:]

test_vector = [5.74, 3.72, 6.28, 8.6, 9.34, 6.47, 2.05, 9.09, 4.39, 4.75]
bubblesort(test_vector)
```

Two things have changed. First, the import statement turned from *ctypes* to *zugbruecke*, although the exact same types, routines and objects were imported. Second, `__bubblesort__` received an additional `memsync` attribute.

Because the `memsync` attribute will be ignored by *ctypes*, you can make the above piece of code platform-independent by adjusting the import statement only. The complete example, which will run on *Unix* and on *Windows* looks just like this:

```
from sys import platform
if any([platform.startswith(os_name) for os_name in ['linux', 'darwin', 'freebsd']]):
        from zugbruecke import windll, cast, pointer, POINTER, c_float, c_int
elif platform.startswith('win'):
        from ctypes import windll, cast, pointer, POINTER, c_float, c_int
else:
        raise # handle other platforms here

__dll__ = windll.LoadLibrary('demo_dll.dll')
__bubblesort__ = __dll__.bubblesort
__bubblesort__.argtypes = (POINTER(c_float), c_int)
__bubblesort__.memsync = [
        {
                'p': [0],
                'l': [1],
                't': 'c_float'
                }
        ]

def bubblesort(values):

        ctypes_float_values = ((c_float)*len(values))(*values)
        ctypes_float_pointer_firstelement = cast(
                pointer(ctypes_float_values), POINTER(c_float)
                )
        __bubblesort__(ctypes_float_pointer_firstelement, len(values))
        values[:] = ctypes_float_values[:]

test_vector = [5.74, 3.72, 6.28, 8.6, 9.34, 6.47, 2.05, 9.09, 4.39, 4.75]
bubblesort(test_vector)
```

5.3 A more complex example: Computing the size of the memory from multiple arguments

There are plenty of cases where you will encounter function (or structure) definitions like the following:

```
void __stdcall __declspec(dllimport) process_image(
        float *image_data,
```

(continues on next page)

(continued from previous page)

```
        int image_width,
        int image_height
        );
```

The `image_data` parameter is a flattened 1D array representing a 2D image. Its length is defined by its width and its height. So the length of the array equals `image_width * image_height`. For cases like this, `memsync` has the ability to dynamically compute the length of the memory through custom functions. Let's have a look at how the above function would be configured in *Python*:

```
process_image.argtypes = (ctypes.POINTER(ctypes.c_float), ctypes.c_int, ctypes.c_int)
process_image.memsync = [
        {
                'p': [0],
                'l': ([1], [2]),
                'f': 'lambda x, y: x * y',
                't': 'c_float'
                }
        ]
```

The above definition will extract the values of the `image_width` and `image_height` parameters for every function call and feed them into the specified lambda function.

5.4 Using string buffers, null-terminated strings and Unicode

Let's assume you are confronted with a regular *Python* (3) string. With the help of a DLL function, you want to replace all occurrences of a letter with another letter.

```
some_string = 'zategahuba'
```

The DLL function's definition looks like this:

```
void __stdcall __declspec(dllimport) replace_letter(
        char *in_string,
        char old_letter,
        char new_letter
        );
```

In *Python*, it can be configured as follows:

```
replace_letter.argtypes = (
        ctypes.POINTER(ctypes.c_char),
        ctypes.c_char,
        ctypes.c_char
        )
replace_letter.memsync = [
        {
                'p': [0],
                'n': True
                }
        ]
```

The above configuration indicates that the first argument of the function is a pointer to a NULL-terminated string.

While *Python* (3) strings are actually Unicode strings, the function accepts an array of type `char` - a bytes array in *Python* terms. I.e. you have to encode the string before it is copied into a string buffer. The following example

illustrates how the function `replace_letter` can be called on the string `some_string`, exchanging all letters `a` with `e`. Subsequently, the result is printed.

```
string_buffer = ctypes.create_string_buffer(some_string.encode('utf-8'))
replace_letter(string_buffer, 'a'.encode('utf-8'), 'e'.encode('utf-8'))
print(string_buffer.value.decode('utf-8'))
```

The process differs if the DLL function accepts Unicode strings. Let's assume the DLL function is defined as follows:

```
void __stdcall __declspec(dllimport) replace_letter_w(
        wchar_t *in_string,
        wchar_t old_letter,
        wchar_t new_letter
        );
```

In Python, it can be configured like this:

```
replace_letter_w.argtypes = (
        ctypes.POINTER(ctypes.c_wchar),
        ctypes.c_wchar,
        ctypes.c_wchar
        )
replace_letter_w.memsync = [
        {
                'p': [0],
                'n': True,
                'w': True
                }
        ]
```

One key aspect has changed: `memsync` contains another field, `w`. It must be set to `True`, indicating that the argument is a Unicode string. Now you can call the function as follows:

```
unicode_buffer = ctypes.create_unicode_buffer(some_string)
replace_letter_w(unicode_buffer, 'a', 'e')
print(unicode_buffer.value)
```

5.5 Applying memory synchronization to callback functions (function pointers)

Let's assume that you're dealing with structures of the following kind:

```
class image_data(ctypes.Structure):
        _fields_ = [
                ('data', ctypes.POINTER(ctypes.c_int16)),
                ('width', ctypes.c_int16),
                ('height', ctypes.c_int16)
                ]
```

2D monochrome image data is represented as a flattened 1D array, field `data`, with size information attached to it in the fields `width` and `height`. You furthermore have a function prototype which accepts an `image_data` structure as an argument:

```
filter_func_type = ctypes.WINFUNCTYPE(ctypes.c_int16, ctypes.POINTER(image_data))
```

Before you actually decorate a *Python* function with it, all you have to do is to change the contents of the `memsync` attribute of the function prototype, `filter_func_type`:

```
filter_func_type.memsync = [
        {
                'p': [0, 'data'],
                'l': ([0, 'width'], [0, 'height']),
                'f': 'lambda x, y: x * y',
                't': 'c_int16'
                }
        ]
```

The above syntax also does not interfere with `ctypes` on *Windows*, i.e. the code remains perfectly platform-independent. Once the function prototype has been configured through `memsync`, it can be applied to a *Python* function:

```
@filter_func_type
def filter_edge_detection(in_buffer):
        # do something ...
```

5.6 Attribute: `memsync` (list of dict)

`memsync` is a list of dictionaries. Every dictionary represents one memory section, which must be kept in sync. It has the following keys:

- `p` (*path to pointer*)
- `l` (*path to length*, optional)
- `n` (*NULL-terminated string flag*, optional)
- `w` (*Unicode character flag*, optional)
- `t` (*data type of pointer*, optional)
- `f` (*custom length function*, optional)
- `_c` (*custom data type*, optional)

5.6.1 Key: `p`, path to pointer (list of int and/or str)

This parameter describes where in the arguments or return value (along the lines of `argtypes` and `restype`) *zugbruecke*'s parser can find the pointer, which it is expected to handle. Consider the following example:

```
# arg nr:    0        1        2
some_routine(param_a, param_b, param_c)
```

If `param_b` was the pointer, `p` would be `[1]` (a list with a single int), referring to the second argument of `some_routine` (counted from zero).

The following more complex example illustrates why `p` is a list actually representing something like a "path":

```
class some_struct(Structure):
        _fields_ = [
                ('field_a', POINTER(c_float)),
                ('field_b', c_int)
```

(continues on next page)

(continued from previous page)

```
                ]

# arg nr:           0        1        2        3
some_other_routine(param_a, param_b, param_c, param_d)
```

Let's assume that `param_a` is of type `some_struct` and `field_a` contains the pointer. `p` would look like this: `[0, 'field_a']`. The pointer is found in `field_a` of the first parameter of `some_other_routine`, `param_a`.

Return values or elements within can be targeted by setting the first element of a path to `'r'` (instead of an integer targeting an argument).

5.6.2 Key: `l`, path to length (list of int and/or str OR tuple of lists of int and/or str) (optional)

This parameter works just like the *path to pointer* parameter. It is expected to tell the parser, where it can find a number (int) which represents the length of the memory block or, alternatively, arguments for a custom length function.

It is expected to be either a single path list like `[0, 'field_a']` or a tuple of multiple (or even zero) path lists, if the optional `f` key (custom length function) is defined.

5.6.3 Key: `n`, NULL-terminated string flag (optional)

Can be set to `True` if a NULL-terminated string is passed as an argument. `memsync` will automatically determine the length of the string, so no extra information on its length (through `l` is required).

5.6.4 Key: `w`, Unicode character flag (optional)

If a Unicode string (buffer) is passed into a function, this parameter must be set to `True`. If not specified, it will default to `False`.

5.6.5 Key: `t`, data type of pointer (PyCSimpleType or PyCStructType) (optional)

This field expects a string representing the name of a ctypes datatype. If you want to specify a custom structure type, you simply specify its class name as a string instead.

This parameter will be used by `ctypes.sizeof` for determining the datatype's size in bytes. The result is then multiplied with the `length` to get an actual size of the memory block in bytes. If it is not explicitly defined, it defaults to `'c_ubyte'`.

For details on `sizeof`, consult the Python documentation on sizeof. It will accept fundamental types as well as structure types.

5.6.6 Key: `f`, custom function for computing the length of the memory segment (optional)

This field can be used to plug in a string, which can be parsed into a function or lambda expression for computing the `length` of the memory section from multiple parameters. The function is expected to accept a number of arguments equal to the number of elements of the tuple of length paths defined in `l`.

5.6.7 Key: _c, custom data type (optional)

If you are using a custom non-*ctypes* datatype, which offers a `from_param` method, you must specify it here. This applies when you construct your own array types or use *numpy* types for instance.

CHAPTER 6

Configuration

zugbruecke can configure itself or can be configured with files or can be (re-) configured during run-time. The configuration allows you to fine-tune its verbosity, architecture and other relevant parameters on a per-session basis.

During import, the *zugbruecke* module starts a default session which is referenced as `zugbruecke.current_session`. Alternatively, you can create your own sessions with `zugbruecke.session()`. See the *chapter on the session model* for details.

6.1 Configuring the session constructor

If you chose to start your own session with `zugbruecke.session()`, the session constructor can be provided with a dictionary containing parameters.

6.2 Configuration files

zugbuecke uses `JSON` configuration files named `.zugbruecke.json`. They are expected in the following locations (in that order):

- The current working directory
- A directory specified in the `ZUGBRUECKE` environment variable
- The *zugbuecke* root directory (`~/.zugbruecke` by default)
- `/etc/zugbruecke`

Parameters passed directly into the *zugbuecke* session constructor will always be given priority. Beyond that, missing parameters are being looked for location after location in the above listed places. If, after checking for configuration files in all those locations, there are still parameters left undefined, *zugbuecke* will fill them with its defaults. A parameter found in a location higher in the list will always be given priority over a the same parameter with different content found in a location further down the list.

6.3 Re-configuration during run-time

Every session exposes a `set_parameter` method, which accepts a dictionary containing parameters.

6.4 Configurable parameters

6.4.1 `id` (str)

Every *zugbuecke* `session` has a unique `id`, which allows easier debugging and keeping track of multiple simultaneously running sessions. If no session id is provided by the user, *zugbuecke* will automatically generate a random hash `string` for every new session. Only manually configure this if absolutely necessary.

6.4.2 `stdout` (bool)

Tells *zugbuecke* to show messages its sub-processes are writing to `stdout`. `True` by default.

6.4.3 `stderr` (bool)

Tells *zugbuecke* to show messages its sub-processes are writing to `stderr`. `True` by default.

6.4.4 `log_write` (bool)

Tells *zugbuecke* to write its logs to disk into the current working directory. `False` by default.

6.4.5 `log_level` (int)

Changes the verbosity of *zugbuecke*. `0` for no logs, `10` for maximum logs. `0` by default.

6.4.6 `arch` (str)

Defines the architecture of *Wine* & *Wine Python*. It can be set to `win32` or `win64`. Default is `win32`, even on 64-bit systems. It appears to be a more stable configuration.

6.4.7 `version` (str)

The `version` parameter tells *zugbuecke* what version of the *Windows CPython* interpreter it should use. By default, it is set to `3.5.3`.

Please note that 3.4 and earlier are not supported. In the opposite direction, at the time of writing, 3.6 (and later) does not work under *Wine* due to a bug in Wine.

6.4.8 `dir` (str)

This parameter defines the root directory of *zugbruecke*. This is where *zugbruecke*'s own *Wine* profile folder is stored (`WINEPREFIX`) and where the *Wine Python environment* resides. By default, it is set to `~/.zugbruecke`.

CHAPTER 7

Platform interoperability

Leveraging the features of *Wine*, *zugbruecke* tries to make things as easy as possible for the user. Some issues remain, though, which must be handled manually by the user. *zugbruecke* offers special APIs for this purpose.

7.1 Module: `zugbruecke.wine`

7.1.1 Method: `path_unix_to_wine`

Parameters:

- `path_in` (str)

Return value

- `path_out` (str)

Converts an absolute or relative *Unix* path into a *Windows* path. It does not check, whether the path actually exists or not. It uses *Wine*'s internal implementation for path conversion.

7.1.2 Method: `path_wine_to_unix`

Parameters:

- `path_in` (str)

Return value

- `path_out` (str)

Converts an absolute or relative *Windows* path into a *Unix* path. It does not check, whether the path actually exists or not. It uses *Wine*'s internal implementation for path conversion.

CHAPTER 8

Wine Python environment

zugbruecke offers a few useful helper scripts for allowing to work with *Python* on *Wine* more easily. They are also used for *zugbruecke*'s internal operations and development tests.

8.1 Command: `wine-python`

This command behaves just like the regular `python` command in a *Unix* shell, except that it fires up a *Windows Python* interpreter on top of *Wine*. It works with all regular parameters and switches, accepts pipes and can be launched in interactive mode.

You can also use it for creating executable *Python* scripts by adding the following at their top: `#!/usr/bin/env wine-python`. Do not forget `chmod +x your_script.py`.

8.2 Command: `wine-pip`

This command behaves just like the regular `pip` command on *Unix*, except that it attempts to install *Python* packages into a dedicated *Python* environment under *Wine*. So if you need any specific packages in `wine-python`, this is how you install them. Most packages written in pure *Python* should work just fine. Anything requiring a compiler during installation does not work. Packages / wheels with pre-compiled binary components in them might work, although this is largely untested territory. Feel free to report any (positive or negative) results.

8.3 Command: `wine-pytest`

This command behaves just like the regular `pytest` or `py.test` command on *Unix*. It is used for verifying how *ctypes* behaves on *Windows* / *Wine*. Every test *zugbruecke* passes when tested with `pytest` is also supposed to be passed by *ctypes* when tested with `wine-pytest`.

CHAPTER 9

Benchmarks

zugbruecke performs reasonably well given its complexity with **less than 0.2 µs overhead per call** in average on modern hardware. It is not (yet) optimized for speed.

The inter-process communication via *multiprocessing connection* adds overhead to every function call. Because *zugbruecke* takes care of packing and unpacking of pointers and structures for arguments and return values, this adds another bit of overhead. Calls are slow in general, but the first call of an individual routine within a session is even slower due to necessary initialization happening beforehand. Depending on the use-case, instead of working with *zugbruecke*, it will be significantly faster to isolate functionality depending on DLL calls into a dedicated *Python* script and run it directly with a *Windows Python* interpreter under *Wine*. *zugbruecke* offers a *Wine Python environment* for this purpose.

For comparison and overhead measurements, see the following numbers:

example call	iterations [#]	w/o zugbruecke [s]	w/ zugbruecke [s]	overhead/call [ns]	parameter features
simple_demo_routine	100k	0.101	11.273	111.7	2x by value
gdc	100k	0.104	11.318	112.1	2x by value
in_mandel (inside)	100k	0.518	11.719	112.0	3x by value
in_mandel (outside)	100k	0.131	11.494	113.6	3x by value
divide	100k	0.174	11.808	116.3	2x by value, 1x by reference
distance	100k	0.230	12.760	125.3	2x struct by reference

Benchmarks were performed with an *i7* 3740QM CPU, *Linux* kernel 4.4.72, *Wine* 2.10, *CPython* 3.6.1 x86-64 for *Linux* and *CPython* 3.5.3 x86-32 for *Windows*. *zugbruecke* was *configured* with log level 0 (logs off) for minimal overhead.

For the corresponding DLL source code (written in C) check the demo_dll directory of this project. For the corresponding Python code check the examples directory of this project.

CHAPTER 10

Security

zugbruecke is **notoriously insecure by design**.

- **DO NOT** run it on any system directly exposed to the internet! Have a firewall on at all times!
- **DO NOT** run untrusted code (or DLLs)!
- **DO NOT** use *zugbruecke* for any security related tasks such as encryption, decryption, authentication and handling of keys or passwords!
- **DO NOT** run it with root / super users privileges!

The following problems also directly apply to *zugbruecke*:

- *Wine* can in fact theoretically run (some) Windows malware.
- **NEVER run Wine as root!** See FAQ at WineHQ for details.

zugbruecke does not actively prohibit its use with root privileges.

CHAPTER 11

Bugs

Please report bugs in *zugbruecke* in *zugbruecke*'s GitHub issue tracker.

Please report bugs in *ctypes* in the Python tracker.

Please report bugs in *Wine* in the WineHQ Bug Tracking System.

Make sure to separate between *zugbruecke*-related, *ctypes*-related and *Wine*-related bugs.

11.1 How to bisect issues

zugbruecke is based on a *session model*. Each session can be launched with parameters, which can either be passed into or picked up from a configuration file by the session constructor. It is also possible to change parameters during run-time.

If you want to increase the log level during run-time, you can do the following:

```
import zugbruecke
# work with zugbruecke
zugbruecke.current_session.set_parameter({'log_level': 10})
# proceed as usual - with a lot more verbosity
```

Alternatively, you can drop a configuration file named `.zugbruecke.json` into your current working directory or *zugbruecke*'s configuration directory (likely `~/.zugbruecke`) and add configuration parameters to it, for example:

```
{"log_level": 10, "log_write": true}
```

The higher the log level, the more output you will get. Default is 0 for no logs. The on-screen log is color-coded for readability. The log can also, in addition, be written to disk, where every log item with plenty of meta data is represented as a one-line JSON object for easy parsing and analysis of larger log files.

For more configuration options check the *chapter on configuration*.

As an alternative approach, you can also check what happens if you run your code directly in a *Windows Python* interpreter with *ctypes*. Consult the *chapter on the Wine Python environment* for details. It is easy to get *ctypes* syntax wrong, so this is a good approach for getting it right.

If in doubt, please also test your code with *ctypes* on an actual Windows system - it might be a bug in this module as well.

CHAPTER 12

FAQ

12.1 Why? Seriously, why?

Good question. Have a look at the *motivation* section in the introduction.

12.2 What are actual use cases for this project?

Read the section on *use cases* in the introduction.

12.3 How does it work?

Have a closer look at the section describing the *implementation* in the introduction.

12.4 Is it secure?

No. See *chapter on security*.

12.5 How fast/slow is it?

It performs reasonably well. See *benchmark section*.

12.6 Can it handle structures?

Yes, in principle. Though, limitations apply. See next question for details.

12.7 Can it handle pointers?

Yes and no.

Pointers to simple C data types (int, float, etc.) used as function parameters or within structures can be handled just fine.

Pointers to arbitrary data structures can be handled if another parameter of the call contains the length of the memory section the pointer is pointing to. *zugbruecke* uses a special `memsync` protocol for indicating which memory sections must be kept in sync between the *Unix* and the *Wine* side of the code. If run on *Windows*, the regular *ctypes* will just ignore any `memsync` directive in the code.

Pointers returned by a DLL pointing to memory allocated by the DLL are currently not handled.

12.8 Is it thread-safe?

Probably (yes). More extensive tests are required.

If you want to be on the safe side, start one *zugbruecke* session per thread in your code manually. You can do this as follows:

```
from zugbruecke import session
# start new thread or process (multiprocessing) - then, inside, do:
a = session()
# now you can do stuff like
kernel32 = a.load_library('kernel32', 'cdll')
# do not forget to terminate the session (i.e. the Windows Python interpreter)
a.terminate()
```

Interested in contributing?

CHAPTER 13

Indices and tables

- genindex
- search

Index