## Contents

1. **What's New** ........................................ 3
  1.1 v1.4.2 ............................................. 3
  1.2 v1.4.1 ............................................. 3
  1.3 v1.4.0 ............................................. 3
  1.4 v1.3.1 ............................................. 4
  1.5 v1.3.0 ............................................. 4
  1.6 v1.2.1 ............................................. 4
  1.7 v1.2.0 ............................................. 4
  1.8 v1.1.2 ............................................. 5
  1.9 v1.1.1 ............................................. 5
  1.10 v1.1.0 ........................................... 5
  1.11 v1.0.3 ........................................... 6
  1.12 v1.0.2 ........................................... 6
  1.13 v1.0.1 ........................................... 6
  1.14 v1.0.0 ........................................... 6

2. **Installation and Status** ........................................ 7
  2.1 Platform-specific instructions ........................................ 8

3. **Overview** ........................................ 11
  3.1 Simple example (ABI level, in-line) ........................................ 11
  3.2 Out-of-line example (ABI level, out-of-line) ........................................ 12
  3.3 Real example (API level, out-of-line) ........................................ 13
  3.4 Struct/Array Example (minimal, in-line) ........................................ 14
  3.5 Purely for performance (API level, out-of-line) ........................................ 14
  3.6 What actually happened? ........................................ 15
  3.7 ABI versus API ........................................ 15

4. **Using the ffi/lib objects** ........................................ 17
  4.1 Working with pointers, structures and arrays ........................................ 17
  4.2 Python 3 support ........................................ 20
  4.3 An example of calling a main-like thing ........................................ 20
  4.4 Function calls ........................................ 21
  4.5 Variadic function calls ........................................ 22
  4.6 Extern “Python” (new-style callbacks) ........................................ 22
  4.7 Callbacks (old style) ........................................ 26
  4.8 Windows: calling conventions ........................................ 27
  4.9 FFI Interface ........................................ 28
4.10 Reference: conversions .................................................. 32

5 Preparing and Distributing modules ........................................ 35
  5.1 ffi.cdef(): declaring types and functions ............................ 37
  5.2 ffi.dlopen(): loading libraries in ABI mode ....................... 38
  5.3 ffi.set_source(): preparing out-of-line modules .................. 39
  5.4 Letting the C compiler fill the gaps .................................. 39
  5.5 ffi.compile() etc.: compiling out-of-line modules ................. 41
  5.6 ffi.include(): combining multiple CFFI interfaces ............... 41
  5.7 ffi.cdef() limitations .................................................. 42
  5.8 Debugging dlopen’ed C libraries ..................................... 42
  5.9 ffi.verify(): in-line API-mode ....................................... 43
  5.10 Upgrading from CFFI 0.9 to CFFI 1.0 ............................... 44

6 Goals ............................................................................. 47

7 Comments and bugs ............................................................. 49
C Foreign Function Interface for Python. The goal is to provide a convenient and reliable way to call compiled C code from Python using interface declarations written in C.

- **Goals**
  - Comments and bugs
CHAPTER 1

What’s New

1.1 v1.4.2

Nothing changed from v1.4.1.

1.2 v1.4.1

• Fix the compilation failure of cffi on CPython 3.5.0. (3.5.1 works; some detail changed that makes some underscore-starting macros disappear from view of extension modules, and I worked around it, thinking it changed in all 3.5 versions—but no: it was only in 3.5.1.)

1.3 v1.4.0

• A better way to do callbacks has been added (faster and more portable, and usually cleaner). It is a mechanism for the out-of-line API mode that replaces the dynamic creation of callback objects (i.e. C functions that invoke Python) with the static declaration in cdef() of which callbacks are needed. This is more C-like, in that you have to structure your code around the idea that you get a fixed number of function pointers, instead of creating them on-the-fly.
• ffi.compile() now takes an optional verbose argument. When True, distutils prints the calls to the compiler.
• ffi.compile() used to fail if given sources with a path that includes "..". Fixed.
• ffi.init_once() added. See docs.
• dir(lib) now works on libs returned by ffi.dlopen() too.
• Cleaned up and modernized the content of the demo subdirectory in the sources (thanks matti!).
• ffi.new_handle() is now guaranteed to return unique void * values, even if called twice on the same object. Previously, in that case, CPython would return two cdata objects with the same void * value. This change is useful to add and remove handles from a global dict (or set) without worrying about duplicates. It already used to work like that on PyPy. This change can break code that used to work on CPython by relying on the object to be kept alive by other means than keeping the result of ffi.new_handle() alive. (The corresponding warning in the docs of ffi.new_handle() has been here since v0.8!)
1.4 v1.3.1

- The optional typedefs (bool, FILE and all Windows types) were not always available from out-of-line FFI objects.
- Opaque enums are phased out from the cdefs: they now give a warning, instead of (possibly wrongly) being assumed equal to unsigned int. Please report if you get a reasonable use case for them.
- Some parsing details, notably volatile is passed along like const and restrict. Also, older versions of pyccparser mis-parse some pointer-to-pointer types like char * const *: the “const” ends up at the wrong place. Added a workaround.

1.5 v1.3.0

- Added ffi.memmove().
- Pull request #64: out-of-line API mode: we can now declare floating-point types with typedef float... foo_t; This only works if foo_t is a float or a double, not long double.
- Issue #217: fix possible unaligned pointer manipulation, which crashes on some architectures (64-bit, non-x86).
- Issues #64 and #126: when using set_source() or verify(), the const and restrict keywords are copied from the cdef to the generated C code; this fixes warnings by the C compiler. It also fixes corner cases like typedef const int T; T a; which would previously not consider a as a constant. (The cdata objects themselves are never const.)
- Win32: support for __stdcall. For callbacks and function pointers; regular C functions still don’t need to have their calling convention declared.
- Windows: CPython 2.7 distutils doesn’t work with Microsoft’s official Visual Studio for Python, and I’m told this is not a bug. For ffi.compile(), we removed a workaround that was inside cffi but which had unwanted side-effects. Try saying import setuptools first, which patches distutils...

1.6 v1.2.1

Nothing changed from v1.2.0.

1.7 v1.2.0

- Out-of-line mode: int a[][...]; can be used to declare a structure field or global variable which is, simultaneously, of total length unknown to the C compiler (the a[] part) and each element is itself an array of N integers, where the value of N is known to the C compiler (the int and [...] parts around it). Similarly, int a[5][...]; is supported (but probably less useful: remember that in C it means int (a[5])[...]);.
- PyPy: the lib.some_function objects were missing the attributes __name__, __module__ and __doc__ that are expected e.g. by some decorators-management functions from functools.
- Out-of-line API mode: you can now do from _example.lib import x to import the name x from _example.lib, even though the lib object is not a standard module object. (Also works in from _example.lib import *, but this is even more of a hack and will fail if lib happens to declare a name called __all__. Note that * excludes the global variables; only the functions and constants make sense to import like this.)
• `lib.__dict__` works again and gives you a copy of the dict—assuming that `lib` has got no symbol called precisely `__dict__`. (In general, it is safer to use `dir(lib)`.)

• Out-of-line API mode: global variables are now fetched on demand at every access. It fixes issue #212 (Windows DLL variables), and also allows variables that are defined as dynamic macros (like `errno`) or `__thread`-local variables. (This change might also tighten the C compiler’s check on the variables’ type.)

• Issue #209: dereferencing NULL pointers now raises `RuntimeError` instead of segfaulting. Meant as a debugging aid. The check is only for NULL: if you dereference random or dead pointers you might still get segfaults.

• Issue #152: callbacks: added an argument `ffi.callback(..., onerror=...)`. If the main callback function raises an exception and `onerror` is provided, then `onerror(exception, exc_value, traceback)` is called. This is similar to writing a `try: except: in the main callback function, but in some cases (e.g. a signal) an exception can occur at the very start of the callback function—before it had time to enter the `try: except:` block.

• Issue #115: added `ffi.new_allocator()`, which officializes support for alternative allocators.

### 1.8 v1.1.2

• `ffi.gc()`: fixed a race condition in multithreaded programs introduced in 1.1.1

### 1.9 v1.1.1

• Out-of-line mode: `ffi.string()`, `ffi.buffer()` and `ffi.getwinerror()` didn’t accept their arguments as keyword arguments, unlike their in-line mode equivalent. (It worked in PyPy.)

• Out-of-line ABI mode: documented a restriction of `ffi.dlopen()` when compared to the in-line mode.

• `ffi.gc()`: when called several times with equal pointers, it was accidentally registering only the last destructor, or even none at all depending on details. (It was correctly registering all of them only in PyPy, and only with the out-of-line FFIs.)

### 1.10 v1.1.0

• Out-of-line API mode: we can now declare integer types with `typedef int... foo_t;`. The exact size and signedness of `foo_t` is figured out by the compiler.

• Out-of-line API mode: we can now declare multidimensional arrays (as fields or as globals) with `int n[...] [...]. Before, only the outermost dimension would support the `...` syntax.

• Out-of-line ABI mode: we now support any constant declaration, instead of only integers whose value is given in the cdef. Such “new” constants, i.e. either non-integers or without a value given in the cdef, must correspond to actual symbols in the lib. At runtime they are looked up the first time we access them. This is useful if the library defines `extern const sometype somename;`.

• `ffi.addressof(lib, "func_name")` now returns a regular cdata object of type “pointer to function”. You can use it on any function from a library in API mode (in ABI mode, all functions are already regular cdata objects). To support this, you need to recompile your cffi modules.

• Issue #198: in API mode, if you declare constants of a `struct` type, what you saw from `lib.CONSTANT` was corrupted.
• Issue #196: `ffi.set_source("package._ffi", None)` would incorrectly generate the Python source to `package._ffi.py` instead of `package/_ffi.py`. Also fixed: in some cases, if the C file was in `build/foo.c`, the .o file would be put in `build/build/foo.o`.

1.11 v1.0.3

• Same as 1.0.2, apart from doc and test fixes on some platforms.

1.12 v1.0.2

• Variadic C functions (ending in a "..." argument) were not supported in the out-of-line ABI mode. This was a bug—there was even a (non-working) example doing exactly that!

1.13 v1.0.1

• `ffi.set_source()` crashed if passed a `sources=[..]` argument. Fixed by chrippa on pull request #60.
• Issue #193: if we use a struct between the first `cdef()` where it is declared and another `cdef()` where its fields are defined, then this definition was ignored.
• Enums were buggy if you used too many "..." in their definition.

1.14 v1.0.0

• The main news item is out-of-line module generation:
  – for ABI level, with `ffi.dlopen()`
  – for API level, which used to be with `ffi.verify()`, now deprecated
• (this page will list what is new from all versions from 1.0.0 forward.)
CHAPTER 2

Installation and Status

Quick installation for CPython (cffi is distributed with PyPy):

- `pip install cffi`
- or get the source code via the Python Package Index.

In more details:

This code has been developed on Linux, but should work on any POSIX platform as well as on Windows 32 and 64. (It relies occasionally on libffi, so it depends on libffi being bug-free; this may not be fully the case on some of the more exotic platforms.)

CFFI supports CPython 2.6, 2.7, 3.x (tested with 3.2 to 3.4); and is distributed with PyPy (CFFI 1.0 is distributed with and requires PyPy 2.6).

The core speed of CFFI is better than ctypes, with import times being either lower if you use the post-1.0 features, or much higher if you don’t. The wrapper Python code you typically need to write around the raw CFFI interface slows things down on CPython, but not unreasonably so. On PyPy, this wrapper code has a minimal impact thanks to the JIT compiler. This makes CFFI the recommended way to interface with C libraries on PyPy.

Requirements:

- CPython 2.6 or 2.7 or 3.x, or PyPy (PyPy 2.0 for the earliest versions of CFFI; or PyPy 2.6 for CFFI 1.0).
- in some cases you need to be able to compile C extension modules; refer to the appropriate docs for your OS. This includes installing CFFI from sources; or developing code based on `ffi.set_source()` or `ffi.verify()`; or installing such 3rd-party modules from sources.
- on CPython, on non-Windows platforms, you also need to install `libffi-dev` in order to compile CFFI itself.
- `pycparser` ≥ 2.06: [https://github.com/eliben/pycparser](https://github.com/eliben/pycparser) (automatically tracked by `pip install cffi`).
- `py.test` is needed to run the tests of CFFI itself.

Download and Installation:

- [http://pypi.python.org/packages/source/c/cffi/cffi-1.4.2.tar.gz](http://pypi.python.org/packages/source/c/cffi/cffi-1.4.2.tar.gz)
  - MD5: 81357fe5042d00650b85b728cc181df2
  - SHA: 76cff6f11f5b9c6c8e2cfa8bf90b5c944394
- Or grab the most current version from the Bitbucket page: `hg clone https://bitbucket.org/cffi/cffi`
- `python setup.py install` or `python setup_base.py install` (should work out of the box on Linux or Windows; see below for MacOS X or Windows 64.)
• running the tests: `py.test c/ testing/` (if you didn’t install cffi yet, you need first `python setup_base.py build_ext -f -i`)

Demos:

• The demo directory contains a number of small and large demos of using cffi.
• The documentation below might be sketchy on details; for now the ultimate reference is given by the tests, notably testing/cffi/test_verify1.py and testing/cffi0/backend_tests.py.

### 2.1 Platform-specific instructions

**libffi** is notoriously messy to install and use — to the point that CPython includes its own copy to avoid relying on external packages. CFFI does the same for Windows, but not for other platforms (which should have their own working libffi’s). Modern Linuxes work out of the box thanks to `pkg-config`. Here are some (user-supplied) instructions for other platforms.

#### 2.1.1 MacOS X

**Homebrew** (Thanks David Griffin for this)

1. Install homebrew: [http://brew.sh](http://brew.sh)
2. Run the following commands in a terminal

```bash
brew install pkg-config libffi
PKG_CONFIG_PATH=/usr/local/opt/libffi/lib/pkgconfig pip install cffi
```

Alternatively, on OS/X 10.6 (Thanks Juraj Sukop for this)

For building libffi you can use the default install path, but then, in `setup.py` you need to change:

```python
include_dirs = []
```
to:

```python
include_dirs = ['/usr/local/lib/libffi-3.0.11/include']
```

Then running `python setup.py build` complains about “fatal error: error writing to -: Broken pipe”, which can be fixed by running:

```bash
ARCHFLAGS="-arch i386 -arch x86_64" python setup.py build
```

as described [here](http).

#### 2.1.2 Windows (regular 32-bit)

Win32 works and is tested at least each official release.

The recommended C compiler compatible with Python 2.7 is this one: [http://www.microsoft.com/en-us/download/details.aspx?id=44266](http://www.microsoft.com/en-us/download/details.aspx?id=44266) There is a known problem with distutils on Python 2.7, as explained in [https://bugs.python.org/issue23246](https://bugs.python.org/issue23246), and the same problem applies whenever you want to run compile() to build a dll with this specific compiler suite download. `import setuptools` might help, but YMMV

2.1.3 Windows 64

Win64 received very basic testing and we applied a few essential fixes in cffi 0.7. The comment above applies for Python 2.7 on Windows 64 as well. Please report any other issue.

Note as usual that this is only about running the 64-bit version of Python on the 64-bit OS. If you’re running the 32-bit version (the common case apparently), then you’re running Win32 as far as we’re concerned.
CFFI can be used in one of four modes: “ABI” versus “API” level, each with “in-line” or “out-of-line” preparation (or compilation).

The **ABI mode** accesses libraries at the binary level, whereas the **API mode** accesses them with a C compiler. This is described in detail below. In the **in-line mode**, everything is set up every time you import your Python code. In the **out-of-line mode**, you have a separate step of preparation (and possibly C compilation) that produces a module which your main program can then import.

(The examples below assume that you have installed CFFI.)

### 3.1 Simple example (ABI level, in-line)

```python
>>> from cffi import FFI
>>> ffi = FFI()
>>> ffi.cdef(""
... int printf(const char *format, ...);  // copy-pasted from the man page
... ")
>>> C = ffi.dlopen(None)                # loads the entire C namespace
>>> arg = ffi.new("char\[]", "world")  # equivalent to C code: char arg[] = "world";
>>> C.printf("hi there, %s.\n", arg)   # call printf
hi there, world.
17  # this is the return value

Note that on Python 3 you need to pass byte strings to char * arguments. In the above example it would be b"world" and b"hi there, %s!\n". In general it is somestring.encode(myencoding).

**This example does not call any C compiler.**
3.2 Out-of-line example (ABI level, out-of-line)

In a real program, you would not include the `ffi.cdef()` in your main program’s modules. Instead, you can rewrite it as follows. It massively reduces the import times, because it is slow to parse a large C header. It also allows you to do more detailed checkings during build-time without worrying about performance (e.g. calling `cdef()` many times with small pieces of declarations, based on the version of libraries detected on the system).

*This example does not call any C compiler:*

```
# file "simple_example_build.py"

# Note: this particular example fails before version 1.0.2
# because it combines variadic function and ABI level.

from cffi import FFI

ffi = FFI()
ffi.set_source("_simple_example", None)
ffi.cdef("int printf(const char *format, ...);
"")

if __name__ == "__main__":
    ffi.compile()
```

Running it once produces `_simple_example.py`. Your main program only imports this generated module, not `simple_example_build.py` any more:

```
from _simple_example import ffi

lib = ffi.dlopen(None)       # Unix: open the standard C library
# import ctypes.util # or, try this on Windows:
# lib = ffi.dlopen(ctypes.util.find_library("c"))

lib.printf(b"hi there, number %d\n", ffi.cast("int", 2))
```

Note that this `ffi.dlopen()`, unlike the one from in-line mode, does not invoke any additional magic to locate the library: it must be a path name (with or without a directory), as required by the C `dlopen()` or `LoadLibrary()` functions. This means that `ffi.dlopen("libfoo.so")` is ok, but `ffi.dlopen("foo")` is not. In the latter case, you could replace it with `ffi.dlopen(ctypes.util.find_library("foo"))`. Also, `None` is only recognized on Unix to open the standard C library.

For distribution purposes, remember that there is a new `_simple_example.py` file generated. You can either include it statically within your project’s source files, or, with `setuptools`, you can say in the `setup.py`:

```
from setuptools import setup

setup(
    ...
    setup_requires=["cffi>=1.0.0"],
    cffi_modules=["simple_example_build.py:ffi"],
    install_requires=["cffi>=1.0.0"],
    )
```
3.3 Real example (API level, out-of-line)

```python
from cffi import FFI
ffi = FFI()

ffi.set_source("_example",
    """ // passed to the real C compiler
    #include <sys/types.h>
    #include <pwd.h>
    """,
    libraries=[], # or a list of libraries to link with
    # (more arguments like setup.py’s Extension class:
    # include_dirs=[], extra_objects=[], and so on)
)

ffi.cdef("""
    // some declarations from the man page
    struct passwd {
        char *pw_name;
        ...; // literally dot-dot-dot
    };
    struct passwd *getpwuid(int uid);
    """)

if __name__ == "__main__":
    ffi.compile()
```

You need to run the `example_build.py` script once to generate “source code” into the file `example.c` and compile this to a regular C extension module. (CFFI selects either Python or C for the module to generate based on whether the second argument to `set_source()` is `None` or not.)

You need a C compiler for this single step. It produces a file called e.g. `example.so` or `example.pyd`. If needed, it can be distributed in precompiled form like any other extension module.

Then, in your main program, you use:

```python
from _example import ffi, lib
p = lib.getpwuid(0)
assert ffi.string(p.pw_name) == b'root'
```

Note that this works independently of the exact C layout of `struct passwd` (it is “API level”, as opposed to “ABI level”). It requires a C compiler in order to run `example_build.py`, but it is much more portable than trying to get the details of the fields of `struct passwd` exactly right. Similarly, we declared `getpwuid()` as taking an `int` argument. On some platforms this might be slightly incorrect—but it does not matter.

To integrate it inside a `setup.py` distribution with Setuptools:

```python
from setuptools import setup

setup(
    ...
    setup_requires=["cffi>=1.0.0"],
    cffi_modules=["example_build.py:ffi"],
    install_requires=["cffi>=1.0.0"],
)
```

3.3. Real example (API level, out-of-line)
3.4 Struct/Array Example (minimal, in-line)

```python
from cffi import FFI
ffi = FFI()
ffi.cdef('''
    typedef struct {
        unsigned char r, g, b;
    } pixel_t;
'''
)
image = ffi.new("pixel_t[]", 800*600)

f = open('data', 'rb')  # binary mode -- important
f.readinto(ffi.buffer(image))
f.close()

image[100].r = 255
image[100].g = 192
image[100].b = 128

f = open('data', 'wb')
f.write(ffi.buffer(image))
f.close()
```

This can be used as a more flexible replacement of the struct and array modules. You could also call `ffi.new("pixel_t[600][800]")` and get a two-dimensional array.

This example does not call any C compiler.

This example also admits an out-of-line equivalent. It is similar to Out-of-line example (ABI level, out-of-line) above, but without any call to `ffi.dlopen()`. In the main program, you write `from _simple_example import ffi` and then the same content as the in-line example above starting from the line `image = ffi.new("pixel_t[]", 800*600)
```

3.5 Purely for performance (API level, out-of-line)

A variant of the section above where the goal is not to call an existing C library, but to compile and call some C function written directly in the build script:

```python
# file "example_build.py"

from cffi import FFI
ffi = FFI()

ffi.cdef("int foo(int *, int *, int);")

ffi.set_source("_example",
    "
        static int foo(int *buffer_in, int *buffer_out, int x)
        {
            /* some algorithm that is seriously faster in C than in Python */
        }
    "")

if __name__ == "__main__":
    ffi.compile()
```

14 Chapter 3. Overview
# file "example.py"

```python
from _example import ffi, lib
buffer_in = ffi.new("int[]", 1000)
# initialize buffer_in here...

# easier to do all buffer allocations in Python and pass them to C,
# even for output-only arguments
buffer_out = ffi.new("int[]", 1000)

result = lib.foo(buffer_in, buffer_out, 1000)
```

You need a C compiler to run example_build.py, once. It produces a file called e.g. _example.so or _example.pyd. If needed, it can be distributed in precompiled form like any other extension module.

## 3.6 What actually happened?

The CFFI interface operates on the same level as C - you declare types and functions using the same syntax as you would define them in C. This means that most of the documentation or examples can be copied straight from the man pages.

The declarations can contain **types, functions, constants** and **global variables**. What you pass to the `cdef()` must not contain more than that; in particular, `#ifdef` or `#include` directives are not supported. The `cdef` in the above examples are just that - they declared “there is a function in the C level with this given signature”, or “there is a struct type with this shape”.

In the ABI examples, the `dlopen()` calls load libraries manually. At the binary level, a program is split into multiple namespaces—a global one (on some platforms), plus one namespace per library. So `dlopen()` returns a `<FFILibrary>` object, and this object has got as attributes all function, constant and variable symbols that are coming from this library and that have been declared in the `cdef()`. If you have several interdependent libraries to load, you would call `cdef()` only once but `dlopen()` several times.

By opposition, the API mode works more closely like a C program: the C linker (static or dynamic) is responsible for finding any symbol used. You name the libraries in the `libraries` keyword argument to `set_source()`, but never need to say which symbol comes from which library. Other common arguments to `set_source()` include `library_dirs` and `include_dirs`; all these arguments are passed to the standard distutils/setuptools.

The `ffi.new()` lines allocate C objects. They are filled with zeroes initially, unless the optional second argument is used. If specified, this argument gives an “initializer”, like you can use with C code to initialize global variables.

The actual `lib.*()` function calls should be obvious: it’s like C.

## 3.7 ABI versus API

Accessing the C library at the binary level (“ABI”) is fraught with problems, particularly on non-Windows platforms. You are not meant to access fields by guessing where they are in the structures. The C libraries are typically meant to be used with a C compiler.

The “real example” above shows how to do that: this example uses `set_source(..., "C source...")` and never `dlopen()`. When using this approach, we have the advantage that we can use literally “...” at various places in the `cdef()`, and the missing information will be completed with the help of the C compiler. Actually, a single C source file is produced, which contains first the “C source” part unmodified, followed by some “magic” C code and declarations derived from the `cdef()`. When this C file is compiled, the resulting C extension module will contain
all the information we need—or the C compiler will give warnings or errors, as usual e.g. if we misdeclare some function’s signature.

Note that the “C source” part from `set_source()` can contain arbitrary C code. You can use this to declare some more helper functions written in C. To export these helpers to Python, put their signature in the `cdef()` too. (You can use the `static` C keyword in the “C source” part, as in `static int myhelper(int x) { return x * 42; }`, because these helpers are only referenced from the “magic” C code that is generated afterwards in the same C file.)

This can be used for example to wrap “crazy” macros into more standard C functions. The extra layer of C can be useful for other reasons too, like calling functions that expect some complicated argument structures that you prefer to build in C rather than in Python. (On the other hand, if all you need is to call “function-like” macros, then you can directly declare them in the `cdef()` as if they were functions.)

The generated piece of C code should be the same independently on the platform on which you run it (or the Python version), so in simple cases you can directly distribute the pre-generated C code and treat it as a regular C extension module. The special Setuptools lines in the example above are meant for the more complicated cases where we need to regenerate the C sources as well—e.g. because the Python script that regenerates this file will itself look around the system to know what it should include or not. Note that the “API level + in-line” mode combination is deprecated. It used to be done with `lib = ffi.verify("C header")`. The out-of-line variant with `set_source("modname", "C header")` is preferred.
Using the ffi/lib objects

4.1 Working with pointers, structures and arrays

The C code’s integers and floating-point values are mapped to Python’s regular `int`, `long` and `float`. Moreover, the C type `char` corresponds to single-character strings in Python. (If you want it to map to small integers, use either `signed char` or `unsigned char`.)

Similarly, the C type `wchar_t` corresponds to single-character unicode strings. Note that in some situations (a narrow Python build with an underlying 4-bytes `wchar_t` type), a single `wchar_t` character may correspond to a pair of surrogates, which is represented as a unicode string of length 2. If you need to convert such a 2-chars unicode string to an integer, `ord(x)` does not work; use instead `int(ffi.cast('wchar_t', x`).

Pointers, structures and arrays are more complex: they don’t have an obvious Python equivalent. Thus, they correspond to objects of type `cdata`, which are printed for example as `<cdata 'struct foo_s *' 0xa3290d8`.

`ffi.new(cctype, [initializer])`: this function builds and returns a new `cdata` object of the given `cctype`. The `cctype` is usually some constant string describing the C type. It must be a pointer or array type. If it is a pointer, e.g. "int *" or `struct foo *`, then it allocates the memory for one `int` or `struct foo`. If it is an array, e.g. `int[10]`, then it allocates the memory for ten `int`. In both cases the returned `cdata` is of type `cctype`.

The memory is initially filled with zeros. An initializer can be given too, as described later.
Example:

```python
>>> ffi.new("int *")
<cdatal int * owning 4 bytes>
>>> ffi.new("int[10]"
<cdatal int[10] owning 40 bytes>

>>> ffi.new("char *")  # allocates only one char---not a C string!
<cdatal char * owning 1 bytes>

>>> ffi.new("char[]", "foobar")  # this allocates a C string, ending in \0
<cdatal char[] owning 7 bytes>
```

Unlike C, the returned pointer object has ownership on the allocated memory: when this exact object is garbage-collected, then the memory is freed. If, at the level of C, you store a pointer to the memory somewhere else, then make sure you also keep the object alive for as long as needed. (This also applies if you immediately cast the returned pointer to a pointer of a different type: only the original object has ownership, so you must keep it alive. As soon as you forget it, then the casted pointer will point to garbage! In other words, the ownership rules are attached to the wrapper cdata objects: they are not, and cannot, be attached to the underlying raw memory.) Example:

```python
global_weakkeydict = weakref.WeakKeyDictionary()

def make_foo():
    s1 = ffi.new("struct foo *")
    fld1 = ffi.new("struct bar *")
    fld2 = ffi.new("struct bar *")
    s1.thefield1 = fld1
    s1.thefield2 = fld2
    # here the 'fld1' and 'fld2' object must not go away,
    # otherwise 's1.thefield1/2' will point to garbage!
    global_weakkeydict[s1] = (fld1, fld2)
    # now 's1' keeps alive 'fld1' and 'fld2'. When 's1' goes
    # away, the weak dictionary entry will be removed.
    return s1
```

The cdata objects support mostly the same operations as in C: you can read or write from pointers, arrays and structures. Dereferencing a pointer is done usually in C with the syntax *p, which is not valid Python, so instead you have to use the alternative syntax p[0] (which is also valid C). Additionally, the p.x and p->x syntaxes in C both become p.x in Python.

We have ffi.NULL to use in the same places as the C NULL. Like the latter, it is actually defined to be ffi.cast("void *", 0). For example, reading a NULL pointer returns a <cdatal type *' NULL>, which you can check for e.g. by comparing it with ffi.NULL.

There is no general equivalent to the & operator in C (because it would not fit nicely in the model, and it does not seem to be needed here). But see ffi.addressof().

Any operation that would in C return a pointer or array or struct type gives you a fresh cdata object. Unlike the “original” one, these fresh cdata objects don’t have ownership: they are merely references to existing memory.

As an exception to the above rule, dereferencing a pointer that owns a struct or union object returns a cdata struct or union object that “co-owns” the same memory. Thus in this case there are two objects that can keep the same memory alive. This is done for cases where you really want to have a struct object but don’t have any convenient place to keep alive the original pointer object (returned by ffi.new()).

Example:

```python
# void somefunction(int *);

x = ffi.new("int *")  # allocate one int, and return a pointer to it
x[0] = 42  # fill it
```
The equivalent of C casts are provided with `ffi.cast("type", value)`. They should work in the same cases as they do in C. Additionally, this is the only way to get cdata objects of integer or floating-point type:

```
>>> x = ffi.cast("int", 42)
>>> x
<cdata 'int' 42>
>>> int(x)
42
```

To cast a pointer to an int, cast it to `intptr_t` or `uintptr_t`, which are defined by C to be large enough integer types (example on 32 bits):

```
>>> int(ffi.cast("intptr_t", pointer_cdata))  # signed
-1340782304
>>> int(ffi.cast("uintptr_t", pointer_cdata))  # unsigned
2954184992L
```

The initializer given as the optional second argument to `ffi.new()` can be mostly anything that you would use as an initializer for C code, with lists or tuples instead of using the C syntax `{ ..., ... }`. Example:

```python
typedef struct { int x, y; } foo_t;

foo_t v = { 1, 2 };  # C syntax
v = ffi.new("foo_t *", [1, 2])  # CFFI equivalent

foo_t v = { .y=1, .x=2 };  # C99 syntax
v = ffi.new("foo_t *", {'y': 1, 'x': 2})  # CFFI equivalent
```

Like C, arrays of chars can also be initialized from a string, in which case a terminating null character is appended implicitly:

```
>>> x = ffi.new("char[]", "hello")
>>> x
<cdata 'char[]' owning 6 bytes>
>>> len(x)  # the actual size of the array
6
>>> x[5]  # the last item in the array
'\x00'
>>> x[0] = 'H'  # change the first item
'Hello'
```

Similarly, arrays of wchar_t can be initialized from a unicode string, and calling `ffi.string()` on the cdata object returns the current unicode string stored in the wchar_t array (adding surrogates if necessary).

Note that unlike Python lists or tuples, but like C, you **cannot** index in a C array from the end using negative numbers.

More generally, the C array types can have their length unspecified in C types, as long as their length can be derived from the initializer, like in C:

```c
int array[] = { 1, 2, 3, 4 };  // C syntax
array = ffi.new("int[]", [1, 2, 3, 4])  # CFFI equivalent
```

As an extension, the initializer can also be just a number, giving the length (in case you just want zero-initialization):

```c
int array[1000];  // C syntax
array = ffi.new("int[1000]")  # CFFI 1st equivalent
array = ffi.new("int[]", 1000)  # CFFI 2nd equivalent
```
This is useful if the length is not actually a constant, to avoid things like `ffi.new("int[%d]", x)`. Indeed, this is not recommended: `ffi` normally caches the string "int[]" to not need to re-parse it all the time.

The C99 variable-sized structures are supported too, as long as the initializer says how long the array should be:

```c
# typedef struct { int x; int y[]; } foo_t;

p = ffi.new("foo_t *"); # length 3
p = ffi.new("foo_t *"); [5, [6, 7, 8]] # length 3 with 0 in the array
p = ffi.new("foo_t *"); [5, 3] # length 3 with 0 everywhere
```

Finally, note that any Python object used as initializer can also be used directly without `ffi.new()` in assignments to array items or struct fields. In fact, `p = ffi.new("T*", initializer) is equivalent to p = ffi.new("T*"), p[0] = initializer.Examples:

```python
# if 'p' is a <cdata 'int[5][5]'>
p[2] = [10, 20] # writes to p[2][0] and p[2][1]

# if 'p' is a <cdata 'foo_t *'>, and foo_t has fields x, y and z
p[0] = {'x': 10, 'y': 20} # writes to p.x and p.z; p.y unmodified

# if, on the other hand, foo_t has a field 'char a[5]':
p.a = "abc" # writes 'a', 'b', 'c' and '\0'; p.a[4] unmodified
```

In function calls, when passing arguments, these rules can be used too; see *Function calls*.

## 4.2 Python 3 support

Python 3 is supported, but the main point to note is that the `char` C type corresponds to the `bytes` Python type, and not `str`. It is your responsibility to encode/decode all Python strings to bytes when passing them to or receiving them from CFFI.

This only concerns the `char` type and derivative types; other parts of the API that accept strings in Python 2 continue to accept strings in Python 3.

## 4.3 An example of calling a main-like thing

Imagine we have something like this:

```python
from cffi import FFI
ffi = FFI()
ffi.cdef(""
    int main_like(int argv, char *argv[]);
""
)
lib = ffi.dlopen("some_library.so")
```

Now, everything is simple, except, how do we create the `char**` argument here? The first idea:

```python
lib.main_like(2, ["arg0", "arg1"])
```

does not work, because the initializer receives two Python `str` objects where it was expecting `<cdata 'char *'>` objects. You need to use `ffi.new()` explicitly to make these objects:

```python
lib.main_like(2, [ffi.new("char[]", "arg0{}"),
                 ffi.new("char[]", "arg1{}")])
```
Note that the two `<cdata 'char[]'>` objects are kept alive for the duration of the call: they are only freed when the list itself is freed, and the list is only freed when the call returns.

If you want instead to build an “argv” variable that you want to reuse, then more care is needed:

```c
# DOES NOT WORK!
argv = ffi.new("char *["], [ffi.new("char[]", "arg0"),
                        ffi.new("char[]", "arg1")])
```

In the above example, the inner “arg0” string is deallocated as soon as “argv” is built. You have to make sure that you keep a reference to the inner “char[]” objects, either directly or by keeping the list alive like this:

```c
argv_keepalive = [ffi.new("char[]", "arg0"),
                 ffi.new("char[]", "arg1")]
argv = ffi.new("char *["], argv_keepalive)
```

### 4.4 Function calls

When calling C functions, passing arguments follows mostly the same rules as assigning to structure fields, and the return value follows the same rules as reading a structure field. For example:

```c
# int foo(short a, int b);

n = lib.foo(2, 3)  # returns a normal integer
lib.foo(40000, 3)  # raises OverflowError
```

You can pass to `char *` arguments a normal Python string (but don’t pass a normal Python string to functions that take a `char *` argument and may mutate it!):

```c
# size_t strlen(const char *);

assert lib.strlen("hello") == 5
```

You can also pass unicode strings as `wchar_t *` arguments. Note that in general, there is no difference between C argument declarations that use `type *` or `type[]`. For example, `int *` is fully equivalent to `int[]` (or even `int[5]; the 5 is ignored). So you can pass an `int *` as a list of integers:

```c
# void do_something_with_array(int *array);

lib.do_something_with_array([1, 2, 3, 4, 5])
```

See [Reference: conversions](#reference-conversions) for a similar way to pass `struct foo_s *` arguments—but in general, it is clearer to simply pass `ffi.new('struct foo_s *', initializer).`

CFFI supports passing and returning structs to functions and callbacks. Example:

```c
# struct foo_s { int a, b; }
# struct foo_s function_returning_a_struct(void);

myfoo = lib.function_returning_a_struct()
```

There are a few (obscure) limitations to the argument types and return type. You cannot pass directly as argument a union (but a `pointer` to a union is fine), nor a struct which uses bitfields (but a `pointer` to such a struct is fine). If you pass a struct (not a `pointer` to a struct), the struct type cannot have been declared with “...;” in the `cdef();` you need to declare it completely in `cdef();`. You can work around these limitations by writing a C function with a simpler signature in the C header code passed to `ffi.set_source();`, and have this C function call the real one.
Aside from these limitations, functions and callbacks can receive and return structs.

For performance, API-level functions are not returned as `<cdata>` objects, but as a different type (on CPython, `<built-in function>`). This means you cannot e.g. pass them to some other C function expecting a function pointer argument. Only `ffi.typeof()` works on them. To get a cdata containing a regular function pointer, use `ffi.addressof(lib, "name")` (new in version 1.1).

Before version 1.1 (or with the deprecated `ffi.verify()`), if you really need a cdata pointer to the function, use the following workaround:

```c
define("int (*foo)(int a, int b);")
```

i.e. declare them as pointer-to-function in the cdef (even if they are regular functions in the C code).

### 4.5 Variadic function calls

Variadic functions in C (which end with “...” as their last argument) can be declared and called normally, with the exception that all the arguments passed in the variable part must be cdata objects. This is because it would not be possible to guess, if you wrote this:

```c
lib.printf("hello, %d\n", 42)  // doesn't work!
```

that you really meant the 42 to be passed as a C `int`, and not a `long` or long long. The same issue occurs with float versus double. So you have to force cdata objects of the C type you want, if necessary with `ffi.cast()`:

```c
lib.printf("hello, %d\n", ffi.cast("int", 42))
lib.printf("hello, %ld\n", ffi.cast("long", 42))
lib.printf("hello, %f\n", ffi.cast("double", 42))
```

But of course:

```c
lib.printf("hello, %s\n", ffi.new("char[]", "world"))
```

Note that if you are using `dlopen()`, the function declaration in the `cdef()` must match the original one in C exactly, as usual — in particular, if this function is variadic in C, then its `cdef()` declaration must also be variadic. You cannot declare it in the `cdef()` with fixed arguments instead, even if you plan to only call it with these argument types. The reason is that some architectures have a different calling convention depending on whether the function signature is fixed or not. (On x86-64, the difference can sometimes be seen in PyPy’s JIT-generated code if some arguments are double.)

Note that the function signature `int foo();` is interpreted by CFFI as equivalent to `int foo(void);`. This differs from the C standard, in which `int foo();` is really like `int foo(...);` and can be called with any arguments. (This feature of C is a pre-C89 relic: the arguments cannot be accessed at all in the body of `foo()` without relying on compiler-specific extensions. Nowadays virtually all code with `int foo();` really means `int foo(void);`.)

### 4.6 Extern “Python” (new-style callbacks)

When the C code needs a pointer to a function which invokes back a Python function of your choice, here is how you do it in the out-of-line API mode. The next section about Callbacks describes the ABI-mode solution.

This is new in version 1.4. Use old-style Callbacks if backward compatibility is an issue. (The original callbacks are slower to invoke and have the same issue as libffi’s callbacks; notably, see the warning. The new style described in the present section does not use libffi’s callbacks at all.) In the builder script, declare in the cdef a function prefixed with `extern "Python":`
CFFI Documentation, Release 1.4.2

```c
ffi.cdef(""
extern "Python" int my_callback(int, int);

void library_function(int(*callback)(int, int));
""
```

```c
ffi.set_source("_my_example", ""
#include <some_library.h>
""
```

The function `my_callback()` is then implemented in Python inside your application’s code:

```python
from _my_example import ffi, lib

@ffi.def_extern()
def my_callback(x, y):
    return 42
```

You obtain a `<cdata>` pointer-to-function object by getting `lib.my_callback`. This `<cdata>` can be passed to C code and then works like a callback: when the C code calls this function pointer, the Python function `my_callback` is called. (You need to pass `lib.my_callback` to C code, and not `my_callback`: the latter is just the Python function above, which cannot be passed to C.)

CFFI implements this by defining `my_callback` as a static C function, written after the `set_source()` code. The `<cdata>` then points to this function. What this function does is invoke the Python function object that is, at runtime, attached with `@ffi.def_extern()`.

The `@ffi.def_extern()` decorator should be applied to a global function, once. This is because each function from the `cdef` with `extern "Python"` turns into only one C function. To support some corner cases, it is possible to redefine the attached Python function by calling `@ffi.def_extern()` again—but this is not recommended! Better write the single global Python function more flexibly in the first place. Calling `@ffi.def_extern()` again changes the C logic to call the new Python function; the old Python function is not callable any more and the C function pointer you get from `lib.my_function` is always the same.

### 4.6.1 Extern “Python” and `void *` arguments

As described just before, you cannot use `extern "Python"` to make a variable number of C function pointers. However, achieving that result is not possible in pure C code either. For this reason, it is usual for C to define callbacks with a `void *data` argument. You can use `ffi.new_handle()` and `ffi.from_handle()` to pass a Python object through this `void *` argument. For example, if the C type of the callbacks is:

```c
typedef void (*event_cb_t)(event_t *evt, void *userdata);
```

and you register events by calling this function:

```c
void event_cb_register(event_cb_t cb, void *userdata);
```

Then you would write this in the build script:

```c
ffi.cdef(""
    typedef ... event_t;
    typedef void (*event_cb_t)(event_t *evt, void *userdata);
    void event_cb_register(event_cb_t cb, void *userdata);

    extern "Python" void my_event_callback(event_t *, void *);
""
```

```c
ffi.set_source("_demo_cffi", ""
#include <the_event_library.h>
""
```

### 4.6. Extern “Python” (new-style callbacks)
and in your main application you register events like this:

```python
from _demo_cffi import ffi, lib

class Widget(object):
    def __init__(self):
        userdata = ffi.new_handle(self)
        self._userdata = userdata  # must keep this alive!
        lib.event_cb_register(lib.my_event_callback, userdata)

    def process_event(self, evt):
        ...

@ffi.def_extern()
def my_event_callback(evt, userdata):
    widget = ffi.from_handle(userdata)
    widget.process_event(evt)
```

Some other libraries don’t have an explicit `void *` argument, but let you attach the `void *` to an existing structure. For example, the library might say that `widget->userdata` is a generic field reserved for the application. If the event’s signature is now this:

```plaintext
typedef void (*event_cb_t)(widget_t *w, event_t *evt);
```

Then you can use the `void *` field in the low-level `widget_t *` like this:

```python
from _demo_cffi import ffi, lib

class Widget(object):
    def __init__(self):
        ll_widget = lib.new_widget(500, 500)
        self.ll_widget = ll_widget  # <cdata 'struct widget *'>
        userdata = ffi.new_handle(self)
        self._userdata = userdata  # must still keep this alive!
        ll_widget.userdata = userdata  # this makes a copy of the "void *"
        lib.event_cb_register(ll_widget, lib.my_event_callback)

    def process_event(self, evt):
        ...

@ffi.def_extern()
def my_event_callback(ll_widget, evt):
    widget = ffi.from_handle(ll_widget.userdata)
    widget.process_event(evt)
```

### 4.6.2 Extern “Python” accessed from C directly

In case you want to access some extern "Python" function directly from the C code written in `set_source()`, you need to write a forward static declaration. The real implementation of this function is added by CFFI after the C code—this is needed because the declaration might use types defined by `set_source()` (e.g. `event_t` above, from the `#include`), so it cannot be generated before.

```c
#include "_demo_cffi.h"

static void my_event_callback(widget_t *, event_t *);
```
This can also be used to write custom C code which calls Python directly. Here is an example (inefficient in this case, but might be useful if the logic in my_algo() is much more complex):

```c
ffi.cdef(""
    extern "Python" int f(int);
    int my_algo(int);
"")
ffi.set_source("_example_cffi", ""
    static int f(int); /* the forward declaration */
    static int my_algo(int n) {
        int i, sum = 0;
        for (i = 0; i < n; i++)
            sum += f(i); /* call f() here */
        return sum;
    }
"")
```

### 4.6.3 Extern “Python”: reference

`extern "Python"` must appear in the `cdef()`. Like the C++ `extern "C"` syntax, it can also be used with braces around a group of functions:

```c
extern "Python" {
    int foo(int);
    int bar(int);
}
```

The `extern "Python"` functions cannot be variadic for now. This may be implemented in the future.

Each corresponding Python callback function is defined with the `@ffi.def_extern()` decorator. Be careful when writing this function: if it raises an exception, or tries to return an object of the wrong type, then the exception cannot be propagated. Instead, the exception is printed to stderr and the C-level callback is made to return a default value. This can be controlled with `error` and `onerror`, described below.

The `@ffi.def_extern()` decorator takes these optional arguments:

- **name**: the name of the function as written in the `cdef`. By default it is taken from the name of the Python function you decorate.

- **error**: the returned value in case the Python function raises an exception. It is 0 or null by default. The exception is still printed to stderr, so this should be used only as a last-resort solution.

- **onerror**: if you want to be sure to catch all exceptions, use `@ffi.def_extern(onerror=my_handler)`. If an exception occurs and `onerror` is specified, then `onerror(exception, exc_value, traceback)` is called. This is useful in some situations where you cannot simply write `try: except:` in the main callback function, because it might not catch exceptions raised by signal handlers: if a signal occurs while in C, the Python signal handler is called as soon as possible, which is after entering the callback function but before executing even the `try:`. If the signal handler raises, we are not in the `try: except:` yet.

    If `onerror` is called and returns normally, then it is assumed that it handled the exception on its own and nothing is printed to stderr. If `onerror` raises, then both tracebackes are printed. Finally, `onerror` can itself provide the result value of the callback in C, but doesn’t have to: if it simply returns `None`—or if `onerror` itself fails—then the value of `error` will be used, if any.
Note the following hack: in onerror, you can access the original callback arguments as follows. First check if traceback is not None (it is None e.g. if the whole function ran successfully but there was an error converting the value returned: this occurs after the call). If traceback is not None, then traceback.tb_frame is the frame of the outermost function, i.e. directly the frame of the function decorated with @ffi.def_extern(). So you can get the value of argname in that frame by reading traceback.tb_frame.f_locals[‘argname’].

4.7 Callbacks (old style)

Here is how to make a new <cdata> object that contains a pointer to a function, where that function invokes back a Python function of your choice:

```python
>>> @ffi.callback("int(int, int)"
>>> def myfunc(x, y):
...   return x + y
...  

>>> myfunc
<cdata 'int(*)(int, int)' calling <function myfunc at 0xf757bbc4>>
```

Note that "int(*)(int, int)" is a C function pointer type, whereas "int(int, int)" is a C function type. Either can be specified to ffi.callback() and the result is the same.

**Warning:** Callbacks are provided for the ABI mode or for backward compatibility. If you are using the out-of-line API mode, it is recommended to use the extern “Python” mechanism instead of callbacks: it gives faster and cleaner code. It also avoids a SELinux issue whereby the setting of deny_execmem must be left to off in order to use callbacks. (A fix in cffi was attempted—see the ffi_closure_alloc branch—but was not merged because it creates potential memory corruption with fork(). For more information, see here.)

Warning: like ffi.new(), ffi.callback() returns a cdata that has ownership of its C data. (In this case, the necessary C data contains the libffi data structures to do a callback.) This means that the callback can only be invoked as long as this cdata object is alive. If you store the function pointer into C code, then make sure you also keep this object alive for as long as the callback may be invoked. The easiest way to do that is to always use @ffi.callback() at module-level only, and to pass "context" information around with ffi.new_handle(), if possible. Example:

```python
# a good way to use this decorator is once at global level
@ffi.callback("int(int, void *)")
def my_global_callback(x, handle):
    return ffi.from_handle(handle).some_method(x)

class Foo(object):
    def __init__(self):
        handle = ffi.new_handle(self)
        self._handle = handle  # must be kept alive
        lib.register_stuff_with_callback_and_voidp_arg(my_global_callback, handle)

    def some_method(self, x):
        ...
```

(See also the section about extern “Python” above, where the same general style is used.)

Note that callbacks of a variadic function type are not supported. A workaround is to add custom C code. In the following example, a callback gets a first argument that counts how many extra int arguments are passed:
# file "example_build.py"

```python
import cffi
ffi = cffi.FFI()
ffi.cdef(""
    int (*python_callback)(int how_many, int *values);
    void *const c_callback; /* pass this const ptr to C routines */
"")
lib = ffi.set_source("_example", ""
    #include <stdarg.h>
    #include <alloca.h>
    static int (*python_callback)(int how_many, int *values);
    static int c_callback(int how_many, ...) {
        va_list ap;
        /* collect the "..." arguments into the values[] array */
        int i, *values = alloca(how_many * sizeof(int));
        va_start(ap, how_many);
        for (i=0; i<how_many; i++)
            values[i] = va_arg(ap, int);
        va_end(ap);
        return python_callback(how_many, values);
    }
"")
```

# file "example.py"

```python
from _example import ffi, lib

@ffi.callback("int(int, int *)")
def python_callback(how_many, values):
    print values  # a list
    return 0
lib.python_callback = python_callback
```

Deprecated: you can also use `ffi.callback()` not as a decorator but directly as `ffi.callback("int(int, int)", myfunc)`. This is discouraged: using this a style, we are more likely to forget the callback object too early, when it is still in use.

The `ffi.callback()` decorator also accepts the optional argument `error`, and from CFFI version 1.2 the optional argument `onerror`. These two work in the same way as described above for extern "Python".

### 4.8 Windows: calling conventions

On Win32, functions can have two main calling conventions: either “cdecl” (the default), or “stdcall” (also known as “WINAPI”). There are also other rare calling conventions, but these are not supported. *New in version 1.3.*

When you issue calls from Python to C, the implementation is such that it works with any of these two main calling conventions; you don’t have to specify it. However, if you manipulate variables of type “function pointer” or declare callbacks, then the calling convention must be correct. This is done by writing `__cdecl` or `__stdcall` in the type, like in C:

```python
@ffi.callback("int __stdcall(int, int)")
def AddNumbers(x, y):
    return x + y
```

or:

4.8. Windows: calling conventions 27
__stdcall is supported but is always the default so it can be left out. In the cdef(), you can also use WINAPI as equivalent to __stdcall. As mentioned above, it is not needed (but doesn’t hurt) to say WINAPI or __stdcall when declaring a plain function in the cdef(). (The difference can still be seen if you take explicitly a pointer to this function with ffi.addressof(), or if the function is extern "Python".)

These calling convention specifiers are accepted but ignored on any platform other than 32-bit Windows.

In CFFI versions before 1.3, the calling convention specifiers are not recognized. In API mode, you could work around it by using an indirection, like in the example in the section about Callbacks("example_build.py"). There was no way to use stdcall callbacks in ABI mode.

4.9 FFI Interface

ffi.new(cdecl, init=None): allocate an instance according to the specified C type and return a pointer to it. The specified C type must be either a pointer or an array: new(‘X *’) allocates an X and returns a pointer to it, whereas new(‘X[n]’) allocates an array of n X’es and returns an array referencing it (which works mostly like a pointer, like in C). You can also use new(‘X[]’, n) to allocate an array of a non-constant length n. See above for other valid initializers. When the returned <cdata> object goes out of scope, the memory is freed. In other words the returned <cdata> object has ownership of the value of type cdecl that it points to. This means that the raw data can be used as long as this object is kept alive, but must not be used for a longer time. Be careful about that when copying the pointer to the memory somewhere else, e.g. into another structure.

ffi.cast("C type", value): similar to a C cast: returns an instance of the named C type initialized with the given value. The value is casted between integers or pointers of any type.

ffi.error: the Python exception raised in various cases. (Don’t confuse it with ffi.errno.)

ffi.errno: the value of errno received from the most recent C call in this thread, and passed to the following C call. (This is a read-write property.)

ffi.getwinerror(code=-1): on Windows, in addition to errno we also save and restore the GetLastError() value across function calls. This function returns this error code as a tuple (code, message), adding a readable message like Python does when raising WindowsError. If the argument code is given, format that code into a message instead of using GetLastError(). (Note that it is also possible to declare and call the GetLastError() function as usual.)

ffi.string(cdata, [maxlen]): return a Python string (or unicode string) from the ‘cdata’.

• If ‘cdata’ is a pointer or array of characters or bytes, returns the null-terminated string. The returned string extends until the first null character, or at most ‘maxlen’ characters. If ‘cdata’ is an array then ‘maxlen’ defaults to its length. See ffi.buffer() below for a way to continue past the first null character. Python 3: this returns a bytes, not a str.

• If ‘cdata’ is a pointer or array of wchar_t, returns a unicode string following the same rules.

• If ‘cdata’ is a single character or byte or a wchar_t, returns it as a byte string or unicode string. (Note that in some situation a single wchar_t may require a Python unicode string of length 2.)

• If ‘cdata’ is an enum, returns the value of the enumerator as a string. If the value is out of range, it is simply returned as the stringified integer.
ffi.buffer(cdata, [size]): return a buffer object that references the raw C data pointed to by the given ‘cdata’, of ‘size’ bytes. The ‘cdata’ must be a pointer or an array. If unspecified, the size of the buffer is either the size of what cdata points to, or the whole size of the array. Getting a buffer is useful because you can read from it without an extra copy, or write into it to change the original value.

Here are a few examples of where buffer() would be useful:

- use file.write() and file.readinto() with such a buffer (for files opened in binary mode)
- use ffi.buffer(mystruct[0])[:].tosocket.recv(len(buffer)) to read into a struct over a socket, rewriting the contents of mystruct[0]

Remember that like in C, you can use array + index to get the pointer to the index’th item of an array.

The returned object is not a built-in buffer nor memoryview object, because these objects’ API changes too much across Python versions. Instead it has the following Python API (a subset of Python 2’s buffer):

- buf[:].or bytes(buf): fetch a copy as a regular byte string (or buf[start:end] for a part)
- buf[:] = newstr: change the original content (or buf[start:end] = newstr)
- len(buf), buf[index], buf[index] = newchar: access as a sequence of characters.

The buffer object returned by ffi.buffer(cdata) keeps alive the cdata object: if it was originally an owning cdata, then its owned memory will not be freed as long as the buffer is alive.

Python 2/3 compatibility note: you should avoid using str(buf), because it gives inconsistent results between Python 2 and Python 3. (This is similar to how str() gives inconsistent results on regular byte strings). Use buf[:] instead.

ffi.from_buffer(python_buffer): return a <cdata 'char[]'> that points to the data of the given Python object, which must support the buffer interface. This is the opposite of ffi.buffer(). It gives a reference to the existing data, not a copy; for this reason, and for PyPy compatibility, it does not work with the built-in types str or unicode or bytearray (or buffers/memoryviews on them). It is meant to be used on objects containing large quantities of raw data, like array.array or numpy arrays. It supports both the old buffer API (in Python 2.x) and the new memoryview API. Note that if you pass a read-only buffer object, you still get a regular <cdata 'char[]'>: it is your responsibility not to write there if the original buffer doesn’t expect you to. The original object is kept alive (and, in case of memoryview, locked) as long as the cdata object returned by ffi.from_buffer() is alive. New in version 0.9.

ffi.memmove(dest, src, n): copy n bytes from memory area src to memory area dest. See examples below. Inspired by the C functions memcpy() and memmove()—like the latter, the areas can overlap. Each of dest and src can be either a cdata pointer or a Python object supporting the buffer/memoryview interface. In the case of dest, the buffer/memoryview must be writable. Unlike ffi.from_buffer(), there are no restrictions on the type of buffer. New in version 1.3. Examples:

- ffi.memmove(myptr, b"hello", 5) copies the 5 bytes of b"hello" to the area that myptr points to.
- ba = bytearray(100); ffi.memmove(ba, myptr, 100) copies 100 bytes from myptr into the bytearray ba.
- ffi.memmove(myptr + 1, myptr, 100) shifts 100 bytes from the memory at myptr to the memory at myptr + 1.

ffi.typeof(“C type” or cdata object): return an object of type <ctype> corresponding to the parsed string, or to the C type of the cdata instance. Usually you don’t need to call this function or to explicitly manipulate <ctype> objects in your code: any place that accepts a C type can receive either a string or a pre-parsed ctype object (and because of caching of the string, there is no real performance difference). It can still be useful in writing typechecks, e.g.:

```python
def myfunction(ptr):
    assert ffi.typeof(ptr) is ffi.typeof("foo_t")
...```
Note also that the mapping from strings like "foo_t*" to the <ctype> objects is stored in some internal dictionary. This guarantees that there is only one <ctype 'foo_t *'> object, so you can use the is operator to compare it. The downside is that the dictionary entries are immortal for now. In the future, we may add transparent reclamation of old, unused entries. In the meantime, note that using strings like "int[%d]" % length to name a type will create many immortal cached entries if called with many different lengths.

ffi.CData, ffi.CType: the Python type of the objects referred to as <cdata> and <ctype> in the rest of this document. Note that some cdata objects may be actually of a subclass of ffi.CData, and similarly with ctype, so you should check with if isinstance(x, ffi.CData). Also, <ctype> objects have a number of attributes for introspection: kind and cname are always present, and depending on the kind they may also have item, length, fields, args, result, ellipsis, abi, elements and relements.

ffi.NULL: a constant NULL of type <cdata 'void *'>.

ffi.sizeof("C type" or cdata object): return the size of the argument in bytes. The argument can be either a C type, or a cdata object, like in the equivalent sizeof operator in C.

ffi.alignof("C type"): return the natural alignment size in bytes of the argument. Corresponds to the __alignof__ operator in GCC.

ffi.offsetof("C struct or array type", *fields_or_indexes): return the offset within the struct of the given field. Corresponds to offsetof() in C.

New in version 0.9: You can give several field names in case of nested structures. You can also give numeric values which correspond to array items, in case of a pointer or array type. For example, ffi.offsetof("int[5]", 2) is equal to the size of two integers, as is ffi.offsetof("int *", 2).

ffi.getctype("C type" or <ctype>, extra=""): return the string representation of the given C type. If non-empty, the “extra” string is appended (or inserted at the right place in more complicated cases); it can be the name of a variable to declare, or an extra part of the type like "*" or "[5]". For example ffi.getctype(ffi.typeof(x), "*") returns the string representation of the C type “pointer to the same type than x”; and ffi.getctype("char[80]", "a") == "char a[80]".

ffi.gc(cdata, destructor): return a new cdata object that points to the same data. Later, when this new cdata object is garbage-collected, destructor(old_cdata_object) will be called. Example of usage: ptr = ffi.gc(lib.malloc(42), lib.free). Note that like objects returned by ffi.new(), the returned pointer objects have ownership, which means the destructor is called as soon as this exact returned object is garbage-collected.

Note that this should be avoided for large memory allocations or for limited resources. This is particularly true on PyPy: its GC does not know how much memory or how many resources the returned ptr holds. It will only run its GC when enough memory it knows about has been allocated (and thus run the destructor possibly later than you would expect). Moreover, the destructor is called in whatever thread PyPy is at that moment, which might be a problem for some C libraries. In these cases, consider writing a wrapper class with custom __enter__() and __exit__() methods, allocating and freeing the C data at known points in time, and using it in a with statement. ffi.new_handle(py_object): return a non-NULL cdata of type void * that contains an opaque reference to py_object. You can pass it around to C functions or store it into C structures. Later, you can use ffi.from_handle(p) to retrieve the original python object from a value with the same void * pointer. Calling ffi.from_handle(p) is invalid and will likely crash if the cdata object returned by new_handle() is not kept alive!

(In case you are wondering, this void * is not the PyObject * pointer. This wouldn’t make sense on PyPy anyway.)

The ffi.new_handle() / from_handle() functions conceptually work like this:

- new_handle() returns cdata objects that contains references to the Python objects; we call them collectively the “handle” cdata objects. The void * value in these handle cdata objects are random but unique.
- from_handle(p) searches all live “handle” cdata objects for the one that has the same value p as its void * value. It then returns the Python object referenced by that handle cdata object. If none is found, you get “undefined behavior” (i.e. crashes).
The “handle” cdata object keeps the Python object alive, similar to how `ffi.new()` returns a cdata object that keeps a piece of memory alive. If the handle cdata object itself is not alive any more, then the association void * -> python_object is dead and `from_handle()` will crash.

New in version 1.4: two calls to `new_handle(x)` are guaranteed to return cdata objects with different void * values, even with the same x. This is a useful feature that avoids issues with unexpected duplicates in the following trick: if you need to keep alive the “handle” until explicitly asked to free it, but don’t have a natural Python-side place to attach it to, then the easiest is to `add()` it to a global set. It can later be removed from the set by `global_set.discard(p)`, with p any cdata object whose void * value compares equal. `ffi.addressof(cdata, *fields_or_indexes)`: limited equivalent to the ‘&’ operator in C:

1. `ffi.addressof(<cdata ‘struct-or-union’>)` returns a cdata that is a pointer to this struct or union. The returned pointer is only valid as long as the original cdata object is; be sure to keep it alive if it was obtained directly from `ffi.new()`.

2. `ffi.addressof(<cdata>, field-or-index...)` returns the address of a field or array item inside the given structure or array. In case of nested structures or arrays, you can give more than one field or index to look recursively. Note that `ffi.addressof(array, index)` can also be expressed as `array + index`: this is true both in CFFI and in C, where `&array[index]` is just `array + index`.

3. `ffi.addressof(<library>, "name")` returns the address of the named function or global variable from the given library object. New in version 1.1: for functions, it returns a regular cdata object containing a pointer to the function.

Note that the case 1. cannot be used to take the address of a primitive or pointer, but only a struct or union. It would be difficult to implement because only structs and unions are internally stored as an indirect pointer to the data. If you need a C int whose address can be taken, use `ffi.new("int[1"]`) in the first place; similarly, for a pointer, use `ffi.new("foo_t *[1"]`).

`ffi.dlopen(libpath, [flags])`: opens and returns a “handle” to a dynamic library, as a `<lib>` object. See Preparing and Distributing modules.

`ffi.dlclose(lib)`: explicitly closes a `<lib>` object returned by `ffi.dlopen()`.

`ffi.RLTD_...`: constants: flags for `ffi.dlopen()`. `ffi.new_allocator(alloc=None, free=None, should_clear_after_alloc=True)`: returns a new allocator. An “allocator” is a callable that behaves like `ffi.new()` but uses the provided low-level `alloc` and `free` functions. New in version 1.2.

`alloc()` is invoked with the size as sole argument. If it returns NULL, a MemoryError is raised. Later, if `free` is not None, it will be called with the result of `alloc()` as argument. Both can be either Python function or directly C functions. If only `free` is None, then no free function is called. If both `alloc` and `free` are None, the default alloc/free combination is used. (In other words, the call `ffi.new(*args)` is equivalent to `ffi.new_allocator()(*args)`.)

If `should_clear_after_alloc` is set to False, then the memory returned by `alloc()` is assumed to be already cleared (or you are fine with garbage); otherwise CFFI will clear it. `ffi.init_once(function, tag)`: run `function()` once. The `tag` should be a primitive object, like a string, that identifies the function: `function()` is only called the first time we see the `tag`. The return value of `function()` is remembered and returned by the current and all future `init_once()` with the same `tag`. If `init_once()` is called from multiple threads in parallel, all calls block until the execution of `function()` is done. If `function()` raises an exception, it is propagated and nothing is cached (i.e. `function()` will be called again, in case we catch the exception and try `init_once()` again). New in version 1.4.

Example:

```python
from _xyz_cffi import ffi, lib

def initlib():
    lib.init_my_library()
```

4.9. FFI Interface 31
```python
def make_new_foo():
    ffi.init_once(initlib, "init")
    return lib.make_foo()
```

`init_once()` is optimized to run very quickly if `function()` has already been called. (On PyPy, the cost is zero—the JIT usually removes everything in the machine code it produces.)

**Note:** one motivation for `init_once()` is the CPython notion of “subinterpreters” in the embedded case. If you are using the out-of-line API mode, `function()` is called only once even in the presence of multiple subinterpreters, and its return value is shared among all subinterpreters. The goal is to mimic the way traditional CPython C extension modules have their init code executed only once in total even if there are subinterpreters. In the example above, the C function `init_my_library()` is called once in total, not once per subinterpreter. For this reason, avoid Python-level side-effects in `function()` (as they will only be applied in the first subinterpreter to run); instead, return a value, as in the following example:

```python
def init_get_max():
    return lib.initialize_once_and_get_some_maximum_number()

def process(i):
    if i > ffi.init_once(init_get_max, "max"):
        raise IndexError("index too large!")
    ...
```

### 4.10 Reference: conversions

This section documents all the conversions that are allowed when writing into a C data structure (or passing arguments to a function call), and reading from a C data structure (or getting the result of a function call). The last column gives the type-specific operations allowed.
<table>
<thead>
<tr>
<th>C type</th>
<th>writing into</th>
<th>reading from</th>
<th>other operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>integers and enums</td>
<td>an integer or anything on which int() works (but not a float!). Must be within range.</td>
<td>a Python int or long, depending on the type</td>
<td>int()</td>
</tr>
<tr>
<td>char</td>
<td>a string of length 1 or another &lt;cdata char&gt;</td>
<td>a string of length 1</td>
<td>int()</td>
</tr>
<tr>
<td>wchar_t</td>
<td>a unicode of length 1 (or maybe 2 if surrogates) or another &lt;cdata wchar_t&gt;</td>
<td>a unicode of length 1 (or maybe 2 if surrogates)</td>
<td>int()</td>
</tr>
<tr>
<td>float, double</td>
<td>a float or anything on which float() works</td>
<td>a Python float</td>
<td>float(), int()</td>
</tr>
<tr>
<td>long double</td>
<td>another &lt;cdata&gt; with a long double, or anything on which float() works</td>
<td>a &lt;cdata&gt;, to avoid losing precision (***</td>
<td>float(), int()</td>
</tr>
<tr>
<td>pointers</td>
<td>another &lt;cdata&gt; with a compatible type (i.e. same type or char* or void*, or as an array instead) (*)</td>
<td>a &lt;cdata&gt;</td>
<td>[ ] (****), +, -, bool()</td>
</tr>
<tr>
<td>void *, char *</td>
<td>another &lt;cdata&gt; with any pointer or array type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pointers to structure</td>
<td>same as pointers</td>
<td>[ ], +, -, bool(), and read/write struct fields</td>
<td></td>
</tr>
<tr>
<td>or union</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>function pointers</td>
<td>same as pointers</td>
<td>bool(), call (**</td>
<td></td>
</tr>
<tr>
<td>arrays</td>
<td>a list or tuple of items</td>
<td>a &lt;cdata&gt;</td>
<td></td>
</tr>
<tr>
<td>char[]</td>
<td>same as arrays, or a Python string</td>
<td>len(), iter(), [ ]</td>
<td></td>
</tr>
<tr>
<td>wchar_t[]</td>
<td>same as arrays, or a Python unicode</td>
<td>len(), iter(), [ ], +, -</td>
<td></td>
</tr>
<tr>
<td>structure</td>
<td>a list or tuple or dict of the field values, or a same-type &lt;cdata&gt;</td>
<td>a &lt;cdata&gt;</td>
<td>read/write fields</td>
</tr>
<tr>
<td>union</td>
<td>same as struct, but with at most one field</td>
<td></td>
<td>read/write fields</td>
</tr>
</tbody>
</table>

(*) item * is item[] in function arguments:

In a function declaration, as per the C standard, a item * argument is identical to a item[] argument (and ffi.cdef() doesn’t record the difference). So when you call such a function, you can pass an argument that is accepted by either C type, like for example passing a Python string to a char * argument (because it works for char[] arguments) or a list of integers to a int * argument (it works for int[] arguments). Note that even if you want to pass a single item, you need to specify it in a list of length 1; for example, a struct point_s * argument might be passed as [[x, y]] or [{'x': 5, 'y': 10}].

As an optimization, the CPython version of CFFI assumes that a function with a char * argument to which you pass a Python string will not actually modify the array of characters passed in, and so passes directly a pointer inside the Python string object. (PyPy might in the future do the same, but it is harder because a string object can move in memory when the GC runs.)

(**) C function calls are done with the GIL released.

Note that we assume that the called functions are not using the Python API from Python.h. For example, we don’t check afterwards if they set a Python exception. You may work around it, but mixing CFFI with Python.h is not recommended. (If you do that, on PyPy and on some platforms like Windows, you may need to explicitly link to libpypy-c.dll to access the CPython C API compatibility layer; indeed, CFFI-generated modules on PyPy don’t link to libpypy-c.dll on their own. But really, don’t do that in the first place.)
***long double support:***

We keep long double values inside a cdata object to avoid losing precision. Normal Python floating-point numbers only contain enough precision for a double. If you really want to convert such an object to a regular Python float (i.e. a C double), call float(). If you need to do arithmetic on such numbers without any precision loss, you need instead to define and use a family of C functions like long double add(long double a, long double b);

****Slicing with x[start:stop]:

Slicing is allowed, as long as you specify explicitly both start and stop (and don’t give any step). It gives a cdata object that is a “view” of all items from start to stop. It is a cdata of type “array” (so e.g. passing it as an argument to a C function would just convert it to a pointer to the start item). As with indexing, negative bounds mean really negative indices, like in C. As for slice assignment, it accepts any iterable, including a list of items or another array-like cdata object, but the length must match. (Note that this behavior differs from initialization: e.g. you can say chararray[10:15] = "hello", but the assigned string must be of exactly the correct length; no implicit null character is added.)

***** Enums are handled like ints:

Like C, enum types are mostly int types (unsigned or signed, int or long; note that GCC’s first choice is unsigned). Reading an enum field of a structure, for example, returns you an integer. To compare their value symbolically, use code like if x.field == lib.FOO. If you really want to get their value as a string, use ffi.string(ffi.cast("the_enum_type", x.field)).
Preparing and Distributing modules

There are three or four different ways to use CFFI in a project. In order of complexity:

• The “in-line”, “ABI mode”:

```python
import cffi

ffi = cffi.FFI()
ffi.cdef("C-like declarations")
lib = ffi.dlopen("libpath")

# use ffi and lib here
```

• The “out-of-line”, but still “ABI mode”, useful to organize the code and reduce the import time:

```python
# in a separate file "package/foo_build.py"
import cffi

ffi = cffi.FFI()
ffi.set_source("package._foo", None)
ffi.cdef("C-like declarations")

if __name__ == "__main__":
    ffi.compile()
```

Running `python foo_build.py` produces a file `_foo.py`, which can then be imported in the main program:
from package._foo import ffi
lib = ffi.dlopen("libpath")

# use ffi and lib here

• The “out-of-line”, “API mode” gives you the most flexibility to access a C library at the level of C, instead of at the binary level:

# in a separate file "package/foo_build.py"
import ffi

ffi = cffi.FFI()
ffi.set_source("package._foo", "real C code")  # <=
ffi.cdef("C-like declarations with '...'")

if __name__ == "__main__":
    ffi.compile(verbose=True)

Running python foo_build.py produces a file _foo.c and invokes the C compiler to turn it into a file _foo.so (or _foo.pyd or _foo.dylib). It is a C extension module which can be imported in the main program:

from package._foo import ffi, lib

# no ffi.dlopen()

# use ffi and lib here

• Finally, you can (but don’t have to) use CFFI’s Distutils or Setuptools integration when writing a setup.py. For Distutils (only in out-of-line API mode):

# setup.py (requires CFI to be installed first)
from distutils.core import setup
import foo_build  # possibly with sys.path tricks to find it

setup(
    ...,
    ext_modules=[foo_build.ffi.distutils_extension()],
)

For Setuptools (out-of-line, but works in ABI or API mode; recommended):

# setup.py (with automatic dependency tracking)
from setuptools import setup

setup(
    ...,
    setup_requires=["cffi>=1.0.0"],
    cffi_modules=["package/foo_build.py:ffi"],
    install_requires=["cffi>=1.0.0"],
)

Note that CFFI actually contains two different FFI classes. The page Using the ffi/lib objects describes the common functionality. It is what you get in the from package._foo import ffi lines above. On the other hand, the extended FFI class is the one you get from import cffi; ffi = cffi.FFI(). It has the same functionality (for in-line use), but also the extra methods described below (to prepare the FFI).

The reason for this split of functionality is that a regular program using CFFI out-of-line does not need to import the cffi pure Python package at all. (Internally it still needs __cffi_backend, a C extension module that comes with
CFFI Documentation, Release 1.4.2

CFFI; this is why CFFI is also listed in install_requires=.. above. In the future this might be split into a different PyPI package that only installs _cffi_backend.

Note that a few small differences do exist: notably, from _foo import ffi returns an object of a type written in C, which does not let you add random attributes to it (nor does it have all the underscore-prefixed internal attributes of the Python version). Similarly, the lib objects returned by the C version are read-only, apart from writes to global variables. Also, lib.__dict__ does not work before version 1.2 or if lib happens to declare a name called __dict__ (use instead dir(lib)). The same is true for lib.__class__ before version 1.4.

5.1 ffi.cdef(): declaring types and functions

ffi.cdef(source): parses the given C source. It registers all the functions, types, constants and global variables in the C source. The types can be used immediately in ffi.new() and other functions. Before you can access the functions and global variables, you need to give ffi another piece of information: where they actually come from (which you do with either ffi.dlopen() or ffi.set_source()). The C source is parsed internally (using pycparser). This code cannot contain #include. It should typically be a self-contained piece of declarations extracted from a man page. The only things it can assume to exist are the standard types:

- char, short, int, long, long long (both signed and unsigned)
- float, double, long double
- intN_t, uintN_t (for N=8,16,32,64), intptr_t, uintptr_t, ptdiff_t, size_t, ssize_t
- wchar_t (if supported by the backend)
- _Bool and bool (equivalent). If not directly supported by the C compiler, this is declared with the size of unsigned char.
- FILE. You can declare C functions taking a FILE * argument and call them with a Python file object. If needed, you can also do c_f = ffi.cast("FILE *", fileobj) and then pass around c_f.
- all common Windows types are defined if you run on Windows (DWORD, LPARAM, etc.). Changed in version 0.9: the types TBYTE TCHAR LPCTSTR PCTSTR LPTSTR PTSTR PTBYTE PTCHAR are no longer automatically defined; see ffi.set_unicode().
- New in version 0.9.3: the other standard integer types from stdint.h, like intmax_t, as long as they map to integers of 1, 2, 4 or 8 bytes. Larger integers are not supported.

The declarations can also contain “...” at various places; these are placeholders that will be completed by the compiler. More information about it below in Letting the C compiler fill the gaps.

Note that all standard type names listed above are handled as defaults only (apart from the ones that are keywords in the C language). If your cdef contains an explicit typedef that redefines one of the types above, then the default described above is ignored. (This is a bit hard to implement cleanly, so in some corner cases it might fail, notably with the error Multiple type specifiers with a type tag. Please report it as a bug if it does.)

Multiple calls to ffi.cdef() are possible. Beware that it can be slow to call ffi.cdef() a lot of times, a consideration that is important mainly in in-line mode.

The ffi.cdef() call takes an optional argument packed: if True, then all structs declared within this cdef are “packed”. (If you need both packed and non-packed structs, use several cdefs in sequence.) This has a meaning similar to __attribute__((packed)) in GCC. It specifies that all structure fields should have an alignment of one byte. (Note that the packed attribute has no effect on bit fields so far, which mean that they may be packed differently than on GCC. Also, this has no effect on structs declared with "...; "—more about it later in Letting the C compiler fill the gaps.)

Note that you can use the type-qualifiers const and restrict (but not __restrict or __restrict__) in the cdef(), but this has no effect on the cdata objects that you get at run-time (they are never const). The effect
is limited to knowing if a global variable is meant to be a constant or not. Also, new in version 1.3: when using set_source() or verify(), these two qualifiers are copied from the cdef to the generated C code; this fixes warnings by the C compiler.

Note a trick if you copy-paste code from sources in which there are extra macros (for example, the Windows documentation uses SAL annotations like _In_ or _Out__). These hints must be removed in the string given to cdef(), but it can be done programmatically like this:

```c
ffi.cdef(re.sub(r"\b(_In_|_Inout_|_Out_|_Outptr_)(opt_)?\b", " ",
"""
    DWORD WINAPI GetModuleFileName(
        _In_opt_ HMODULE hModule,
        _Out_ LPTSTR lpFilename,
        _In_ DWORD nSize
    );
"""
)
```

ffi.set_unicode(enabled_flag): Windows: if enabled_flag is True, enable the UNICODE and _UNICODE defines in C, and declare the types TBYTE TCHAR LPCTSTR PCTSTR LPTSTR PTSTR PTBYTE PTCHAR to be (pointers to) wchar_t. If enabled_flag is False, declare these types to be (pointers to) plain 8-bit characters. (These types are not predeclared at all if you don’t call set_unicode().) New in version 0.9.

The reason behind this method is that a lot of standard functions have two versions, like MessageBoxA() and MessageBoxW(). The official interface is MessageBox() with arguments like LPTCSTR. Depending on whether UNICODE is defined or not, the standard header renames the generic function name to one of the two specialized versions, and declares the correct (unicode or not) types.

Usually, the right thing to do is to call this method with True. Be aware (particularly on Python 2) that, afterwards, you need to pass unicode strings as arguments instead of byte strings. (Before cffi version 0.9, TCHAR and friends where hard-coded as unicode, but UNICODE was, inconsistently, not defined by default.)

### 5.2 ffi.dlopen(): loading libraries in ABI mode

ffi.dlopen(libpath, [flags]): this function opens a shared library and returns a module-like library object. Use this when you are fine with the limitations of ABI-level access to the system. In case of doubt, read again ABI versus API in the overview.

You can use the library object to call the functions previously declared by ffi.cdef(), to read constants, and to read or write global variables. Note that you can use a single cdef() to declare functions from multiple libraries, as long as you load each of them with dlopen() and access the functions from the correct one.

The libpath is the file name of the shared library, which can contain a full path or not (in which case it is searched in standard locations, as described in man dlopen), with extensions or not. Alternatively, if libpath is None, it returns the standard C library (which can be used to access the functions of glibc, on Linux).

Let me state it again: this gives ABI-level access to the library, so you need to have all types declared manually exactly as they were while the library was made. No checking is done. Mismatches can cause random crashes.

Note that only functions and global variables live in library objects; the types exist in the ffi instance independently of library objects. This is due to the C model: the types you declare in C are not tied to a particular library, as long as you #include their headers; but you cannot call functions from a library without linking it in your program, as dlopen() does dynamically in C.

For the optional flags argument, see man dlopen (ignored on Windows). It defaults to ffi.RTLD_NOW.

This function returns a “library” object that gets closed when it goes out of scope. Make sure you keep the library object around as long as needed. (Alternatively, the out-of-line FFIs have a method ffi.dlclose(lib).) Note: the old version of ffi.dlopen() from the in-line ABI mode tries to use ctypes.util.find_library()
if it cannot directly find the library. The newer out-of-line ffi.dlopen() no longer does it automatically; it simply passes the argument it receives to the underlying dlopen() or LoadLibrary() function. If needed, it is up to you to use ctypes.util.find_library() or any other way to look for the library's filename. This also means that ffi.dlopen(None) no longer work on Windows; try instead ffi.dlopen(ctypes.util.find_library('c')).

5.3 ffi.set_source(): preparing out-of-line modules

ffi.set_source(module_name, c_header_source, [**keywords...]): prepare the ffi for producing out-of-line an external module called module_name. New in version 1.0.

ffi.set_source() by itself does not write any file, but merely records its arguments for later. It can therefore be called before or after ffi.cdef().

In ABI mode, you call ffi.set_source(module_name, None). The argument is the name (or dotted name inside a package) of the Python module to generate. In this mode, no C compiler is called.

In API mode, the c_header_source argument is a string that will be pasted into the .c file generated. This piece of C code typically contains some #include, but may also contain more, like definitions for custom “wrapper” C functions. The goal is that the .c file can be generated like this:

```c
#include "Python.h"
...c_header_source...
...magic code...
```

where the “magic code” is automatically generated from the cdef(). For example, if the cdef() contains int foo(int x); then the magic code will contain logic to call the function foo() with an integer argument, itself wrapped inside some CPython or PyPy-specific code.

The keywords arguments to set_source() control how the C compiler will be called. They are passed directly to distutils or setuptools and include at least sources, include_dirs, define_macros, undef_macros, libraries, library_dirs, extra_objects, extra_compile_args and extra_link_args. You typically need at least libraries=['foo'] in order to link with libfoo.so or libfoo.so.X.Y, or foo.dll on Windows. The sources is a list of extra .c files compiled and linked together (the file module_name.c shown above is always generated and automatically added as the first argument to sources). See the distutils documentations for more information about the other arguments.

An extra keyword argument processed internally is source_extension, defaulting to ".c". The file generated will be actually called module_name + source_extension. Example for C++ (but note that there are still a few known issues of C-versus-C++ compatibility):

```c
extern "C" {
    int somefunc(int somearg) { return real_cpp_func(somearg); }
}
'', source_extension='.cpp')
```

5.4 Letting the C compiler fill the gaps

If you are using a C compiler (“API mode”), then:
• functions taking or returning integer or float-point arguments can be misdeclared: if e.g. a function is declared by \texttt{cdef()} as taking a \texttt{int}, but actually takes a \texttt{long}, then the C compiler handles the difference.

• other arguments are checked: you get a compilation warning or error if you pass a \texttt{int *} argument to a function expecting a \texttt{long *}.

• similarly, most other things declared in the \texttt{cdef()} are checked, to the best we implemented so far; mistakes give compilation warnings or errors.

Moreover, you can use “...” (literally, dot-dot-dot) in the \texttt{cdef()} at various places, in order to ask the C compiler to fill in the details. These places are:

• structure declarations: any \texttt{struct \{ \} } that ends with “...;” as the last “field” is partial: it may be missing fields and/or have them declared out of order. This declaration will be corrected by the compiler. (But note that you can only access fields that you declared, not others.) Any \texttt{struct} declaration which doesn’t use “...” is assumed to be exact, but this is checked: you get an error if it is not correct.

• \texttt{New in version 1.1:} integer types: the syntax “\texttt{typedef int... foo_t;}” declares the type \texttt{foo_t} as an integer type whose exact size and signedness is not specified. The compiler will figure it out. (Note that this requires \texttt{set_source();} it does not work with \texttt{verify().} The \texttt{int...} can be replaced with \texttt{long...} or \texttt{unsigned long long...} or any other primitive integer type, with no effect. The type will always map to one of (u)\texttt{int(8,16,32,64)}\_t in Python, but in the generated C code, only \texttt{foo_t} is used.

• \texttt{New in version 1.3:} floating-point types: “\texttt{typedef float... foo_t;}” (or equivalently “\texttt{typedef double... foo_t;}”) declares \texttt{foo_t} as a float-or-a-double; the compiler will figure out which it is. Note that if the actual C type is even larger (\texttt{long double} on some platforms), then compilation will fail. The problem is that the Python “float” type cannot be used to store the extra precision. (Use the non-dot-dot-dot syntax \texttt{typedef long double foo_t;} as usual, which returns values that are not Python floats at all but cdata “long double” objects.)

• unknown types: the syntax “\texttt{typedef ... foo_t;}” declares the type \texttt{foo_t} as opaque. Useful mainly for when the API takes and returns \texttt{foo_t *} without you needing to look inside the \texttt{foo_t}. Also works with “\texttt{typedef ... *foo_p;}” which declares the pointer type \texttt{foo_p} without giving a name to the opaque type itself. Note that such an opaque struct has no known size, which prevents some operations from working (mostly like in C). \texttt{You cannot use this syntax to declare a specific type, like an integer type! It declares opaque struct-like types only}. In some cases you need to say that \texttt{foo_t} is not opaque, but just a struct where you don’t know any field; then you would use “\texttt{typedef struct \{ ...; \} foo_t;}”.

• array lengths: when used as structure fields or in global variables, arrays can have an unspecified length, as in “\texttt{int n[...]};”. The length is completed by the C compiler. This is slightly different from “\texttt{int n[]};”, because the latter means that the length is not known even to the C compiler, and thus no attempt is made to complete it. \texttt{New in version 1.1:} support for multidimensional arrays: “\texttt{int n[...][...];}”.

\texttt{New in version 1.2:} “\texttt{int m[]}[...];”, i.e. \ldots can be used in the innermost dimensions without being also used in the outermost dimension. In the example given, the length of the \texttt{m} array is assumed not to be known to the C compiler, but the length of every item (like the sub-array \texttt{m[0]} is always known the C compiler. In other words, only the outermost dimension can be specified as [], both in C and in CFFI, but any dimension can be given as [\ldots] in CFFI.

• enums: if you don’t know the exact order (or values) of the declared constants, then use this syntax: “\texttt{enum foo \{ A, B, C, ... \};}” (with a trailing “...”). The C compiler will be used to figure out the exact values of the constants. An alternative syntax is “\texttt{enum foo \{ A=..., B, C \};}” or even “\texttt{enum foo \{ .A=..., .B=..., .C=... \};}”. Like with structs, an enum without “...” is assumed to be exact, and this is checked.

• integer constants and macros: you can write in the \texttt{cdef} the line “\texttt{#define FOO ...}”, with any macro name \texttt{FOO} but with \ldots as a value. Provided the macro is defined to be an integer value, this value will be available via an attribute of the library object. The same effect can be achieved by writing a declaration \texttt{static}
const int FOO; The latter is more general because it supports other types than integer types (note: the C syntax is then to write the \texttt{const} together with the variable name, as in static char *\texttt{const} FOO;).

Currently, it is not supported to find automatically which of the various integer or float types you need at which place. If a type is named, and an integer type, then use \texttt{typedef int \ldots the\_type\_name;}. In the case of function arguments or return type, when it is a simple integer/float type, it may be misdeclared (if you misdeclare a function \texttt{void f(long)} as \texttt{void f(int)}, it still works, but you have to call it with arguments that fit an int). But it doesn’t work any longer for more complex types (e.g. you cannot misdeclare a \texttt{int *} argument as \texttt{long *}) or in other locations (e.g. a global array \texttt{int a[5];} must not be misdeclared \texttt{long a[5];}). CFFI considers all types listed above as primitive (so \texttt{long long a[5];} and \texttt{int64\_t a[5]} are different declarations).

5.5 \texttt{ffi.compile()} etc.: compiling out-of-line modules

You can use one of the following functions to actually generate the .py or .c file prepared with \texttt{ffi.set\_source()} and \texttt{ffi.cdef()}.

Note that these function won’t overwrite a .py/.c file with exactly the same content, to preserve the mtime. In some cases where you need the mtime to be updated anyway, delete the file before calling the functions.

\texttt{ffi.compile(tmpdir='.', verbose=False)}: explicitly generate the .py or .c file, and (if .c) compile it. The output file is (or are) put in the directory given by tmpdir. In the examples given here, we use if \texttt{__name__ \!= "\_\_main\_\_": ffi.compile()} in the build scripts—if they are directly executed, this makes them rebuild the .py/.c file in the current directory. (Note: if a package is specified in the call to set\_source(), then a corresponding subdirectory of the tmpdir is used.)

New in version 1.4: verbose argument. If True, it prints the usual distutils output, including the command lines that call the compiler. (This parameter might be changed to True by default in a future release.)

\texttt{ffi.emit\_python\_code(filename)}: generate the given .py file (same as \texttt{ffi.compile()} for ABI mode, with an explicitly-named file to write). If you choose, you can include this .py file pre-packaged in your own distributions: it is identical for any Python version (2 or 3).

\texttt{ffi.emit\_c\_code(filename)}: generate the given .c file (for API mode) without compiling it. Can be used if you have some other method to compile it, e.g. if you want to integrate with some larger build system that will compile this file for you. You can also distribute the .c file: unless the build script you used depends on the OS or platform, the .c file itself is generic (it would be exactly the same if produced on a different OS, with a different version of CPython, or with PyPy; it is done with generating the appropriate \#ifdef).

\texttt{ffi.distutils\_extension(tmpdir=\"build\", verbose=True)}: for distutils-based setup.py files. Calling this creates the .c file if needed in the given tmpdir, and returns a distutils.core.Extension instance.

For Setuptools, you use instead the line \texttt{cffi_modules=["path/to/foo\_build.py:ffi"]} in setup.py. This line asks Setuptools to import and use a helper provided by CFFI, which in turn executes the file path/to/foo\_build.py (as with \texttt{execfile()}) and looks up its global variable called ffi. You can also say \texttt{cffi_modules=["path/to/foo\_build.py:maker"]}, where maker names a global function; it is called with no argument and is supposed to return a FFI object.

5.6 \texttt{ffi.include()}: combining multiple CFFI interfaces

\texttt{ffi.include(other\_ffi)}: includes the typedefs, structs, unions, enums and constants defined in another FFI instance. This is meant for large projects where one CFFI-based interface depends on some types declared in a different CFFI-based interface.
Note that you should only use one ffi object per library; the intended usage of ffi.include() is if you want to interface with several inter-dependent libraries. For only one library, make one ffi object. (You can write several cdef() calls over the same ffi from several Python files, if one file would be too large.)

For out-of-line modules, the ffi.include(other_ffi) line should occur in the build script, and the other_ffi argument should be another FFI that comes from another build script. When the two build scripts are turned into generated files, say _ffi.so and _other_ffi.so, then importing _ffi.so will internally cause _other_ffi.so to be imported. At that point, the real declarations from _other_ffi.so are combined with the real declarations from _ffi.so.

The usage of ffi.include() is the cdef-level equivalent of a #include in C, where a part of the program might include types and functions defined in another part for its own usage. You can see on the ffi object (and associated lib objects on the including side) the types and constants declared on the included side. In API mode, you can also see the functions and global variables directly. In ABI mode, these must be accessed via the original other_lib object returned by the dlopen() method on other_ffi.

### 5.7 ffi.cdef() limitations

All of the ANSI C declarations should be supported in cdef(), and some of C99. (This excludes any #include or #ifdef.) Known missing features that are GCC or MSVC extensions:

- Any __attribute__ or #pragma pack(n)
- Additional types: complex numbers, special-size floating and fixed point types, vector types, and so on. You might be able to access an array of complex numbers by declaring it as an array of struct my_complex
  
  { double real, imag; }, but in general you should declare them as struct { ...; } and cannot access them directly. This means that you cannot call any function which has an argument or return value of this type (this would need added support in libffi). You need to write wrapper functions in C, e.g.

  void foo_wrapper(struct my_complex c) { foo(c.real + c.imag*1j); },

  and call foo_wrapper rather than foo directly.
- Function pointers with non-default calling conventions (e.g. on Windows, “stdcall”).

Note that declarations like int field[]; in structures are interpreted as variable-length structures. Declarations like int field[...]; on the other hand are arrays whose length is going to be completed by the compiler. You can use int field[]; for array fields that are not, in fact, variable-length; it works too, but in this case, as CFFI believes it cannot ask the C compiler for the length of the array, you get reduced safety checks: for example, you risk overwriting the following fields by passing too many array items in the constructor.

**New in version 1.2:** Thread-local variables (__thread) can be accessed, as well as variables defined as dynamic macros(#define myvar (*fetchme())). Before version 1.2, you need to write getter/setter functions.

### 5.8 Debugging dlopen’ed C libraries

A few C libraries are actually hard to use correctly in a dlopen() setting. This is because most C libraries are intended for, and tested with, a situation where they are linked with another program, using either static linking or dynamic linking — but from a program written in C, at start-up, using the linker’s capabilities instead of dlopen().

This can occasionally create issues. You would have the same issues in another setting than CFFI, like with ctypes or even plain C code that calls dlopen(). This section contains a few generally useful environment variables (on Linux) that can help when debugging these issues.

export LD_TRACE_LOADED_OBJECTS=all
provides a lot of information, sometimes too much depending on the setting. Output verbose debugging
information about the dynamic linker. If set to all prints all debugging information it has, if set to help
prints a help message about which categories can be specified in this environment variable

export LD_VERBOSE=1

(glibc since 2.1) If set to a nonempty string, output symbol versioning information about the program if
querying information about the program (i.e., either LD_TRACE_LOADED_OBJECTS has been set, or
--list or --verify options have been given to the dynamic linker).

export LD_WARN=1

(ELF only)(glibc since 2.1.3) If set to a nonempty string, warn about unresolved symbols.

5.9 ffi.verify(): in-line API-mode

ffi.verify() is supported for backward compatibility, but is deprecated. ffi.verify(c_header_source,
tmpdir=.., ext_package=.., modulename=.., flags=.., **kwargs) makes and compiles a C
file from the ffi.cdef(), like ffi.set_source() in API mode, and then immediately loads and returns
the dynamic library object. Some non-trivial logic is used to decide if the dynamic library must be recompiled or not; see
below for ways to control it.

The c_header_source and the extra keyword arguments have the same meaning as in ffi.set_source().

One remaining use case for ffi.verify() would be the following hack to find explicitly the size of any type, in
bytes, and have it available in Python immediately (e.g. because it is needed in order to write the rest of the build
script):

```python
ffi = cffi.FFI()
ffi.cdef("const int mysize;")
lib = ffi.verify("const int mysize = sizeof(THE_TYPE);")
print lib.mysize
```

Extra arguments to ffi.verify():

- tmpdir controls where the C files are created and compiled. Unless the CFFI_TMPDIR environment variable
  is set, the default is directory_containing_the_py_file/__pycache__ using the directory name
  of the .py file that contains the actual call to ffi.verify(). (This is a bit of a hack but is generally consistent
  with the location of the .pyc files for your library. The name __pycache__ itself comes from Python 3.)
- ext_package controls in which package the compiled extension module should be looked from. This is only
  useful after distributing ffi.verify()-based modules.
- The tag argument gives an extra string inserted in the middle of the extension module’s name:
  _cffi_<tag>_<hash>. Usefull to give a bit more context, e.g. when debugging.
- The modulename argument can be used to force a specific module name, overriding the name
  _cffi_<tag>_<hash>. Use with care, e.g. if you are passing variable information to verify() but
  still want the module name to be always the same (e.g. absolute paths to local files). In this case, no hash is
  computed and if the module name already exists it will be reused without further check. Be sure to have other
  means of clearing the tmpdir whenever you change your sources.
- source_extension has the same meaning as in ffi.set_source().
- The optional flags argument has been added in version 0.9; see man dlopen (ignored on Windows). It
defaults to ffi.RTLD_NOW. (With ffi.set_source(), you would use sys.setdlopenflags().)
- The optional relative_to argument is useful if you need to list local files passed to the C compiler:
The line above is roughly the same as:

```python
ext = ffi.verify(..., sources=['/path/to/this/file/foo.c'])
```

except that the default name of the produced library is built from the CRC checksum of the argument `sources`, as well as most other arguments you give to `ffi.verify()` — but not `relative_to`. So if you used the second line, it would stop finding the already-compiled library after your project is installed, because the `'/path/to/this/file'` suddenly changed. The first line does not have this problem.

Note that during development, every time you change the C sources that you pass to `cdef()` or `verify()`, then the latter will create a new module file name, based on two CRC32 hashes computed from these strings. This creates more and more files in the `__pycache__` directory. It is recommended that you clean it up from time to time. A nice way to do that is to add, in your test suite, a call to `ffi.verifier.cleanup_tmpdir()`. Alternatively, you can manually remove the whole `__pycache__` directory.

An alternative cache directory can be given as the `tmpdir` argument to `verify()`, via the environment variable `CFFI_TMPDIR`, or by calling `cffi.verifier.set_tmpdir(path)` prior to calling `verify`.

### 5.10 Upgrading from CFFI 0.9 to CFFI 1.0

CFFI 1.0 is backward-compatible, but it is still a good idea to consider moving to the out-of-line approach new in 1.0. Here are the steps.

**ABI mode** if your CFFI project uses `ffi.dlopen()`:

```python
import cffi

ffi = cffi.FFI()
ffi.cdef("stuff")
lib = ffi.dlopen("libpath")
```

and if the “stuff” part is big enough that import time is a concern, then rewrite it as described in the out-of-line but still ABI mode above. Optionally, see also the setuptools integration paragraph. **API mode** if your CFFI project uses `ffi.verify()`:

```python
import cffi

ffi = cffi.FFI()
ffi.cdef("stuff")
lib = ffi.verify("real C code")
```

then you should really rewrite it as described in the out-of-line, API mode above. It avoids a number of issues that have caused `ffi.verify()` to grow a number of extra arguments over time. Then see the distutils or setuptools paragraph. Also, remember to remove the `ext_package=".."` from your `setup.py`, which was sometimes needed with `verify()` but is just creating confusion with `set_source()`. The following example should work both with old (pre-1.0) and new versions of CFFI—supporting both is important to run on PyPy, because CFFI 1.0 does not work in PyPy < 2.6:

```python
# in a separate file "package/foo_build.py"
import cffi

ffi = cffi.FFI()
C_HEADER_SRC = ''
    #include "somelib.h"
    ...
```
C_KEYWORDS = dict(libraries=['somelib'])

if hasattr(ffi, 'set_source'):
    ffi.set_source("package._foo", C_HEADER_SRC, **C_KEYWORDS)

ffi.cdef('''
    int foo(int);
''')

if __name__ == '__main__':
    ffi.compile()

And in the main program:

try:
    from package._foo import ffi, lib
except ImportError:
    from package.foo_build import ffi, C_HEADER_SRC, C_KEYWORDS
    lib = ffi.verify(C_HEADER_SRC, **C_KEYWORDS)

(FWIW, this latest trick can be used more generally to allow the import to “work” even if the _foo module was not generated.)

Writing a setup.py script that works both with CFFI 0.9 and 1.0 requires explicitly checking the version of CFFI that we can have—it is hard-coded as a built-in module in PyPy:

if '_cffi_backend' in sys.builtin_module_names:  # PyPy
    import _cffi_backend
    requires_cffi = "cffi==" + _cffi_backend.__version__
else:
    requires_cffi = "cffi>=1.0.0"

Then we use the requires_cffi variable to give different arguments to setup() as needed, e.g.:

if requires_cffi.startswith("cffi==0.""):
    # backward compatibility: we have "cffi==0.*"
    from package.foo_build import ffi
    extra_args = dict(
        ext_modules=[ffi.verifier.get_extension()],
        ext_packages="...",  # if needed
    )
else:
    extra_args = dict(
        setup_requires=[requires_cffi],
        cffi_modules=['package/foo_build.py:ffi'],
    )

setup(
    name=...,  
    ...,
    install_requires=[requires_cffi],
    **extra_args
)
The interface is based on LuaJIT’s FFI, and follows a few principles:

- The goal is to call C code from Python without learning a 3rd language: existing alternatives require users to learn domain specific language (Cython, SWIG) or API (ctypes). The CFFI design requires users to know only C and Python, minimizing the extra bits of API that need to be learned.

- Keep all the Python-related logic in Python so that you don’t need to write much C code (unlike CPython native C extensions).

- The preferred way is to work at the level of the API (Application Programming Interface): the C compiler is called from the declarations you write to validate and link to the C language constructs. Alternatively, it is also possible to work at the ABI level (Application Binary Interface), the way ctypes work. However, on non-Windows platforms, C libraries typically have a specified C API but not an ABI (e.g. they may document a “struct” as having at least these fields, but maybe more).

- Try to be complete. For now some C99 constructs are not supported, but all C89 should be, including macros (and including macro “abuses”, which you can manually wrap in saner-looking C functions).

- Attempt to support both PyPy and CPython, with a reasonable path for other Python implementations like IronPython and Jython.

- Note that this project is not about embedding executable C code in Python, unlike Weave. This is about calling existing C libraries from Python.

Get started by reading the overview.
Comments and bugs

The best way to contact us is on the IRC #pypy channel of irc.freenode.net. Feel free to discuss matters either there or in the mailing list. Please report to the issue tracker any bugs.

As a general rule, when there is a design issue to resolve, we pick the solution that is the “most C-like”. We hope that this module has got everything you need to access C code and nothing more.

— the authors, Armin Rigo and Maciej Fijalkowski