CFFI Documentation

Release 1.11.2

Armin Rigo, Maciej Fijalkowski

Oct 09, 2017

Contents

C Foreign Function Interface for Python. Interact with almost any C code from Python, based on C-like declarations that you can often copy-paste from header files or documentation.

CHAPTER 1

What's New

v1.11.2

- Fix Windows issue with managing the thread-state on CPython 3.0 to 3.5

v1.11.1

- Fix tests, remove deprecated C API usage
- Fix (hack) for 3.6.0/3.6.1/3.6.2 giving incompatible binary extensions (cpython issue #29943)
- Fix for 3.7.0a1+

v1.11

- Support the modern standard types `char16_t` and `char32_t`. These work like `wchar_t`: they represent one unicode character, or when used as `charN_t *` or `charN_t[]` they represent a unicode string. The difference with `wchar_t` is that they have a known, fixed size. They should work at all places that used to work with `wchar_t` (please report an issue if I missed something). Note that with `set_source()`, you need to make sure that these types are actually defined by the C source you provide (if used in `cdef()`).
- Support the C99 types `float _Complex` and `double _Complex`. Note that libffi doesn't support them, which means that in the ABI mode you still cannot call C functions that take complex numbers directly as arguments or return type.
- Fixed a rare race condition when creating multiple `FFI` instances from multiple threads. (Note that you aren't meant to create many `FFI` instances: in inline mode, you should write `ffi = cffi.FFI()` at module level just after `import cffi`; and in out-of-line mode you don't instantiate `FFI` explicitly at all.)
- Windows: using callbacks can be messy because the CFFI internal error messages show up to stderr—but stderr goes nowhere in many applications. This makes it particularly hard to get started with the embedding mode.

(Once you get started, you can at least use `@ffi.def_extern(onerror=...)` and send the error logs where it makes sense for your application, or record them in log files, and so on.) So what is new in CFFI is that now, on Windows CFFI will try to open a non-modal MessageBox (in addition to sending raw messages to stderr). The MessageBox is only visible if the process stays alive: typically, console applications that crash close immediately, but that is also the situation where stderr should be visible anyway.

- Progress on support for callbacks in NetBSD.
- Functions returning booleans would in some case still return 0 or 1 instead of False or True. Fixed.
- ffi.gc() now takes an optional third parameter, which gives an estimate of the size (in bytes) of the object. So far, this is only used by PyPy, to make the next GC occur more quickly (issue #320). In the future, this might have an effect on CPython too (provided the CPython issue 31105 is addressed).
- Add a note to the documentation: the ABI mode gives function objects that are *slower* to call than the API mode does. For some reason it is often thought to be faster. It is not!

v1.10.1

(only released inside PyPy 5.8.0)

- Fixed the line numbers reported in case of `cdef()` errors. Also, I just noticed, but pycparser always supported the preprocessor directive `# 42 "foo.h"` to mean "from the next line, we're in file foo.h starting from line 42", which it puts in the error messages.

v1.10

- Issue #295: use calloc() directly instead of PyObject_Malloc()+memset() to handle ffi.new() with a default allocator. Speeds up `ffi.new(large-array)` where most of the time you never touch most of the array.
- Some OS/X build fixes ("only with Xcode but without CLT").
- Improve a couple of error messages: when getting mismatched versions of cffi and its backend; and when calling functions which cannot be called with libffi because an argument is a struct that is "too complicated" (and not a struct *pointer*, which always works).
- Add support for some unusual compilers (non-msvc, non-gcc, non-icc, non-clang)
- Implemented the remaining cases for `ffi.from_buffer`. Now all buffer/memoryview objects can be passed. The one remaining check is against passing unicode strings in Python 2. (They support the buffer interface, but that gives the raw bytes behind the UTF16/UCS4 storage, which is most of the times not what you expect. In Python 3 this has been fixed and the unicode strings don't support the memoryview interface any more.)
- The C type `_Bool` or `bool` now converts to a Python boolean when reading, instead of the content of the byte as an integer. The potential incompatibility here is what occurs if the byte contains a value different from 0 and 1. Previously, it would just return it; with this change, CFFI raises an exception in this case. But this case means "undefined behavior" in C; if you really have to interface with a library relying on this, don't use `bool` in the CFFI side. Also, it is still valid to use a byte string as initializer for a `bool[]`, but now it must only contain `\x00` or `\x01`. As an aside, `ffi.string()` no longer works on `bool[]` (but it never made much sense, as this function stops at the first zero).
- `ffi.buffer` is now the name of cffi's buffer type, and `ffi.buffer()` works like before but is the constructor of that type.
- `ffi.addressof(lib, "name")` now works also in in-line mode, not only in out-of-line mode. This is useful for taking the address of global variables.

- Issue #255: `cdata` objects of a primitive type (integers, floats, char) are now compared and ordered by value. For example, `<cdata 'int' 42>` compares equal to `42` and `<cdata 'char' b'A'>` compares equal to `b'A'`. Unlike C, `<cdata 'int' -1>` does not compare equal to `ffi.cast("unsigned int", -1)`: it compares smaller, because `-1 < 4294967295`.
- PyPy: `ffi.new()` and `ffi.new_allocator()()` did not record "memory pressure", causing the GC to run too infrequently if you call `ffi.new()` very often and/or with large arrays. Fixed in PyPy 5.7.
- Support in `ffi.cdef()` for numeric expressions with `+` or `-`. Assumes that there is no overflow; it should be fixed first before we add more general support for arbitrary arithmetic on constants.

v1.9

- Structs with variable-sized arrays as their last field: now we track the length of the array after `ffi.new()` is called, just like we always tracked the length of `ffi.new("int[]", 42)`. This lets us detect out-of-range accesses to array items. This also lets us display a better `repr()`, and have the total size returned by `ffi.sizeof()` and `ffi.buffer()`. Previously both functions would return a result based on the size of the declared structure type, with an assumed empty array. (Thanks andrew for starting this refactoring.)
- Add support in `cdef()/set_source()` for unspecified-length arrays in typedefs: `typedef int foo_t[...];`. It was already supported for global variables or structure fields.
- I turned in v1.8 a warning from `cffi/model.py` into an error: `'enum xxx' has no values explicitly defined: refusing to guess which integer type it is meant to be (unsigned/signed, int/long)`. Now I'm turning it back to a warning again; it seems that guessing that the enum has size `int` is a 99%-safe bet. (But not 100%, so it stays as a warning.)
- Fix leaks in the code handling `FILE *` arguments. In CPython 3 there is a remaining issue that is hard to fix: if you pass a Python file object to a `FILE *` argument, then `os.dup()` is used and the new file descriptor is only closed when the GC reclaims the Python file object—and not at the earlier time when you call `close()`, which only closes the original file descriptor. If this is an issue, you should avoid this automatic convertion of Python file objects: instead, explicitly manipulate file descriptors and call `fdopen()` from C (...via cffi).

v1.8.3

- When passing a `void *` argument to a function with a different pointer type, or vice-versa, the cast occurs automatically, like in C. The same occurs for initialization with `ffi.new()` and a few other places. However, I thought that `char *` had the same property—but I was mistaken. In C you get the usual warning if you try to give a `char *` to a `char **` argument, for example. Sorry about the confusion. This has been fixed in CFFI by giving for now a warning, too. It will turn into an error in a future version.

v1.8.2

- Issue #283: fixed `ffi.new()` on structures/unions with nested anonymous structures/unions, when there is at least one union in the mix. When initialized with a list or a dict, it should now behave more closely like the `{}` syntax does in GCC.

v1.8.1

- CPython 3.x: experimental: the generated C extension modules now use the "limited API", which means that, as a compiled .so/.dll, it should work directly on any version of CPython >= 3.2. The name produced by distutils is still version-specific. To get the version-independent name, you can rename it manually to `NAME.abi3.so`, or use the very recent setuptools 26.
- Added `ffi.compile(debug=...)`, similar to `python setup.py build --debug` but defaulting to True if we are running a debugging version of Python itself.

v1.8

- Removed the restriction that `ffi.from_buffer()` cannot be used on byte strings. Now you can get a `char *` out of a byte string, which is valid as long as the string object is kept alive. (But don't use it to *modify* the string object! If you need this, use `bytearray` or other official techniques.)
- PyPy 5.4 can now pass a byte string directly to a `char *` argument (in older versions, a copy would be made). This used to be a CPython-only optimization.

v1.7

- `ffi.gc(p, None)` removes the destructor on an object previously created by another call to `ffi.gc()`
- `bool(ffi.cast("primitive type", x))` now returns False if the value is zero (including `-0.0`), and True otherwise. Previously this would only return False for cdata objects of a pointer type when the pointer is NULL.
- bytearrays: `ffi.from_buffer(bytearray-object)` is now supported. (The reason it was not supported was that it was hard to do in PyPy, but it works since PyPy 5.3.) To call a C function with a `char *` argument from a buffer object—now including bytearrays—you write `lib.foo(ffi.from_buffer(x))`. Additionally, this is now supported: `p[0:length] = bytearray-object`. The problem with this was that a iterating over bytearrays gives *numbers* instead of *characters*. (Now it is implemented with just a memcpy, of course, not actually iterating over the characters.)
- C++: compiling the generated C code with C++ was supposed to work, but failed if you make use the `bool` type (because that is rendered as the C `_Bool` type, which doesn't exist in C++).
- `help(lib)` and `help(lib.myfunc)` now give useful information, as well as `dir(p)` where `p` is a struct or pointer-to-struct.

v1.6

- ffi.list_types()
- ffi.unpack()
- extern "Python+C"
- in API mode, `lib.foo.__doc__` contains the C signature now. On CPython you can say `help(lib.foo)`, but for some reason `help(lib)` (or `help(lib.foo)` on PyPy) is still useless; I haven't yet figured out the hacks needed to convince `pydoc` to show more. (You can use `dir(lib)` but it is not most helpful.)
- Yet another attempt at robustness of `ffi.def_extern()` against CPython's interpreter shutdown logic.

v1.5.2

- Fix 1.5.1 for Python 2.6.

v1.5.1

- A few installation-time tweaks (thanks Stefano!)
- Issue #245: Win32: `__stdcall` was never generated for `extern "Python"` functions
- Issue #246: trying to be more robust against CPython's fragile interpreter shutdown logic

v1.5.0

- Support for using CFFI for embedding.

v1.4.2

Nothing changed from v1.4.1.

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.)

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!)

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 pycparser mis-parse some pointer-to-pointer types like `char * const *`: the "const" ends up at the wrong place. Added a workaround.

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...

v1.2.1

Nothing changed from v1.2.0.

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.

v1.1.2

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

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.)

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`.

v1.0.3

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

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!

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.

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 (automatically tracked by `pip install cffi`).
- py.test is needed to run the tests of CFFI itself.

Download and Installation:

- https://pypi.python.org/pypi/cffi
- Checksums of the "source" package version 1.11.2:

  - MD5: a731487324b501c8295221b629d3f5f3
  - SHA: 04d2df85eb1921630b4f9206886737eb37200c19
  - SHA256: ab87dd91c0c4073758d07334c1e5f712ce8fe48f007b86f8238773963ee700a6
- 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/cffi1/test_verify1.py and testing/cffi0/backend_tests.py.

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.

MacOS X

Homebrew (Thanks David Griffin for this)

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

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

Aternatively, **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:

```
include_dirs = []
```

to:

```
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:

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

as described here.

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 There is a known problem with distutils on Python 2.7, as explained in 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

For Python 3.4 and beyond: https://www.visualstudio.com/en-us/downloads/visual-studio-2015-ctp-vs

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.

Linux and OS/X: UCS2 versus UCS4

This is about getting an ImportError about `_cffi_backend.so` with a message like `Symbol not found: _PyUnicodeUCS2_AsASCIIString`. This error occurs in Python 2 as soon as you mix "ucs2" and "ucs4" builds of Python. It means that you are now running a Python compiled with "ucs4", but the extension module `_cffi_backend.so` was compiled by a different Python: one that was running "ucs2". (If the opposite problem occurs, you get an error about `_PyUnicodeUCS4_AsASCIIString` instead.)

If you are using `pyenv`, then see https://github.com/yyuu/pyenv/issues/257.

More generally, the solution that should always work is to download the sources of CFFI (instead of a prebuilt binary) and make sure that you build it with the same version of Python than the one that will use it. For example, with virtualenv:

- `virtualenv ~/venv`
- `cd ~/path/to/sources/of/cffi`
- `~/venv/bin/python setup.py build --force` # forcing a rebuild to make sure
- `~/venv/bin/python setup.py install`

This will compile and install CFFI in this virtualenv, using the Python from this virtualenv.

NetBSD

You need to make sure you have an up-to-date version of libffi, which fixes some bugs.

CHAPTER 3

Overview

Contents

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 faster **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.)

Simple example (ABI level, in-line)

```
>>> 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)`.

Python 3 on Windows: `ffi.dlopen(None)` does not work. This problem is messy and not really fixable. The example above could be fixed by calling another function from a specific DLL that exists on your system.

This example does not call any C compiler. It works in the so-called ABI mode, which means that it will crash if you call some function or access some fields of a structure that was slightly misdeclared in the cdef().

If using a C compiler to install your module is an option, it is highly recommended to use the API mode described in the next paragraph. (It is also faster.)

Real example (API level, out-of-line)

```
# file "example_build.py"

# Note: we instantiate the same 'cffi.FFI' class as in the previous
# example, but call the result 'ffibuilder' now instead of 'ffi';
# this is to avoid confusion with the other 'ffi' object you get below

from cffi import FFI
ffibuilder = FFI()

ffibuilder.set_source("_example",
   r""" // 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)

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

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

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:

```
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. It is also faster than the ABI mode.

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

```
from setuptools import setup

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

Struct/Array Example (minimal, in-line)

```
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 *Real example (API level, out-of-line)* above, but passing `None` as the second argument to `ffibuilder.set_source()`. Then 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)`.

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:

```
# file "example_build.py"

from cffi import FFI
ffibuilder = FFI()

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

ffibuilder.set_source("_example",
r"""
    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__":
    ffibuilder.compile(verbose=True)
```

```
# file "example.py"

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.

Out-of-line, ABI level

The out-of-line ABI mode is a mixture of the regular (API) out-of-line mode and the in-line ABI mode. It lets you use the ABI mode, with its advantages (not requiring a C compiler) and problems (crashes more easily).

This mixture mode lets you 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).

```
# file "simple_example_build.py"

from cffi import FFI

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

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

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:ffibuilder"],
    install_requires=["cffi>=1.0.0"],
)
```

Embedding

New in version 1.5.

CFFI can be used for embedding: creating a standard dynamically-linked library (`.dll` under Windows, `.so` elsewhere) which can be used from a C application.

```
import cffi
ffibuilder = cffi.FFI()

ffibuilder.embedding_api("""
    int do_stuff(int, int);
""")
```

```
ffibuilder.set_source("my_plugin", "")

ffibuilder.embedding_init_code("""
    from my_plugin import ffi

    @ffi.def_extern()
    def do_stuff(x, y):
        print("adding %d and %d" % (x, y))
        return x + y
""")

ffibuilder.compile(target="plugin-1.5.*", verbose=True)
```

This simple example creates `plugin-1.5.dll` or `plugin-1.5.so` as a DLL with a single exported function, `do_stuff()`. You execute the script above once, with the interpreter you want to have internally used; it can be CPython 2.x or 3.x or PyPy. This DLL can then be used "as usual" from an application; the application doesn't need to know that it is talking with a library made with Python and CFFI. At runtime, when the application calls `int do_stuff(int, int)`, the Python interpreter is automatically initialized and `def do_stuff(x, y):` gets called. See the details in the documentation about embedding.

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.

ABI versus API

Accessing the C library at the binary level ("ABI") is fraught with problems, particularly on non-Windows platforms.

The most immediate drawback of the ABI level is that calling functions needs to go through the very general *libffi* library, which is slow (and not always perfectly tested on non-standard platforms). The API mode instead compiles a

CPython C wrapper that directly invokes the target function. It is, comparatively, massively faster (and works better than libffi ever can).

The more fundamental reason to prefer the API mode is that *the C libraries are typically meant to be used with a C compiler*. You are not supposed to do things like guess where fields are in the structures. The "real example" above shows how CFFI uses a C compiler under the hood: 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. CFFI will turn this into a single C source file, which contains 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 (which depends on the `_cffi_backend` module, on CPython). 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 exists but is long 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.

CHAPTER 4

Using the ffi/lib objects

Contents

Keep this page under your pillow.

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))`.

New in version 1.11: in addition to `wchar_t`, the C types `char16_t` and `char32_t` work the same but with a known fixed size. In previous versions, this could be achieved using `uint16_t` and `int32_t` but without automatic convertion to Python unicodes.

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(ctype, [initializer])`: this function builds and returns a new cdata object of the given `ctype`. The ctype 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 `ctype`.

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

Example:

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

>>> ffi.new("char *")           # allocates only one char---not a C string!
<cdata 'char *' owning 1 bytes>
>>> ffi.new("char[]", "foobar")  # this allocates a C string, ending in \0
<cdata '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:

```
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, then the weak dictionary entry will be removed.
    return s1
```

Usually you don't need a weak dict: for example, to call a function with a `char * *` argument that contains a pointer to a `char *` pointer, it is enough to do this:

```
p = ffi.new("char[]", "hello, world")     # p is a 'char *'
q = ffi.new("char **", p)                 # q is a 'char **'
lib.myfunction(q)
# p is alive at least until here, so that's fine
```

However, this is always wrong (usage of freed memory):

```
p = ffi.new("char **", ffi.new("char[]", "hello, world"))
# WRONG!  as soon as p is built, the inner ffi.new() gets freed!
```

This is wrong too, for the same reason:

```
p = ffi.new("struct my_stuff")
p.foo = ffi.new("char[]", "hello, world")
# WRONG!  as soon as p.foo is set, the ffi.new() gets freed!
```

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 `<cdata '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). There is ffi.addressof(), but only for some cases. You cannot take the "address" of a number in Python, for example; similarly, you cannot take the address of a CFFI pointer. If you have this kind of C code:

```
int x, y;
fetch_size(&x, &y);

opaque_t *handle;      // some opaque pointer
init_stuff(&handle);   // initializes the variable 'handle'
more_stuff(handle);    // pass the handle around to more functions
```

then you need to rewrite it like this, replacing the variables in C with what is logically pointers to the variables:

```
px = ffi.new("int *")
py = ffi.new("int *")              arr = ffi.new("int[2]")
lib.fetch_size(px, py)    -OR-     lib.fetch_size(arr, arr + 1)
x = px[0]                          x = arr[0]
y = py[0]                          y = arr[1]

p_handle = ffi.new("opaque_t **")
lib.init_stuff(p_handle)    # pass the pointer to the 'handle' pointer
handle = p_handle[0]        # now we can read 'handle' out of 'p_handle'
lib.more_stuff(handle)
```

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:

```
# void somefunction(int *);

x = ffi.new("int *")       # allocate one int, and return a pointer to it
x[0] = 42                  # fill it
lib.somefunction(x)        # call the C function
print x[0]                 # read the possibly-changed value
```

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:

```
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
>>> ffi.string(x) # interpret 'x' as a regular null-terminated string
'Hello'
```

Similarly, arrays of wchar_t or char16_t or char32_t can be initialized from a unicode string, and calling `ffi.string()` on the cdata object returns the current unicode string stored in the source array (adding surrogates if necessary). See the Unicode character types section for more details.

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:

```
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):

```
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:

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

p = ffi.new("foo_t *", [5, [6, 7, 8]]) # length 3
p = ffi.new("foo_t *", [5, 3])         # length 3 with 0 in the array
p = ffi.new("foo_t *", {'y': 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:

```
# 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, 'z': 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*.

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.

An example of calling a main-like thing

Imagine we have something like this:

```
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:

```
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:

```
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:

```
# 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:

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

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:

```
# 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!):

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

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

You can also pass unicode strings as `wchar_t *` or `char16_t *` or `char32_t *` arguments. Note that the C language makes no difference between argument declarations that use `type *` or `type[]`. For example, `int *` is fully equivalent to `int[]` (or even `int[5]`; the 5 is ignored). For CFFI, this means that you can always pass arguments that can be converted to either `int *` or `int[]`. For example:

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

lib.do_something_with_array([1, 2, 3, 4, 5])     # works for int[]
```

See Reference: conversions for a similar way to pass `struct foo_s *` arguments—but in general, it is clearer in this case to pass `ffi.new('struct foo_s *', initializer)`.

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

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

myfoo = lib.function_returning_a_struct()
# `myfoo`: <cdata 'struct foo_s' owning 8 bytes>
```

For performance, non-variadic API-level functions that you get by writing `lib.some_function` are not `<cdata>` objects, but an object of a different type (on CPython, `<built-in function>`). This means you cannot pass them directly 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")`.

There are a few (obscure) limitations to the supported argument and return types. These limitations come from libffi and apply only to calling `<cdata>` function pointers; in other words, they don't apply to non-variadic `cdef()`-declared functions if you are using the API mode. The limitations are that you cannot pass directly as argument or return type:

- a union (but a *pointer* to a union is fine);
- a struct which uses bitfields (but a *pointer* to such a struct is fine);
- a struct that was declared with "`...`" in the `cdef()`.

In API mode, you can work around these limitations: for example, if you need to call such a function pointer from Python, you can instead write a custom C function that accepts the function pointer and the real arguments and that does the call from C. Then declare that custom C function in the `cdef()` and use it from Python.

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:

```
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()`:

```
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:

```
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);`.)

Memory pressure (PyPy)

This paragraph applies only to PyPy, because its garbage collector (GC) is different from CPython's. It is very common in C code to have pairs of functions, one which performs memory allocations or acquires other resources, and the other which frees them again. Depending on how you structure your Python code, the freeing function is only called when the GC decides a particular (Python) object can be freed. This occurs notably in these cases:

- If you use a `__del__()` method to call the freeing function.
- If you use `ffi.gc()`.
- This does not occur if you call the freeing function at a deterministic time, like in a regular `try: finally:` block. It does however occur *inside a generator*— if the generator is not explicitly exhausted but forgotten at a `yield` point, then the code in the enclosing `finally` block is only invoked at the next GC.

In these cases, you may have to use the built-in function `__pypy__.add_memory_pressure(n)`. Its argument `n` is an estimate of how much memory pressure to add. For example, if the pair of C functions that we are talking about is `malloc(n)` and `free()` or similar, you would call `__pypy__.add_memory_pressure(n)` after `malloc(n)`. Doing so is not always a complete answer to the problem, but it makes the next GC occur earlier, which is often enough.

The same applies if the memory allocations are indirect, e.g. the C function allocates some internal data structures. In that case, call `__pypy__.add_memory_pressure(n)` with an argument `n` that is an rough estimation. Knowing the exact size is not important, and memory pressure doesn't have to be manually brought down again after calling the freeing function. If you are writing wrappers for the allocating / freeing pair of functions, you should probably call `__pypy__.add_memory_pressure()` in the former even if the user may invoke the latter at a known point with a `finally:` block.

In case this solution is not sufficient, or if the acquired resource is not memory but something else more limited (like file descriptors), then there is no better way than restructuring your code to make sure the freeing function is called at a known point and not indirectly by the GC.

Note that in PyPy <= 5.6 the discussion above also applies to `ffi.new()`. In more recent versions of PyPy, both `ffi.new()` and `ffi.new_allocator()()` automatically account for the memory pressure they create. (In case you need to support both older and newer PyPy's, try calling `__pypy__.add_memory_pressure()` anyway; it is better to overestimate than not account for the memory pressure.)

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"`:

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

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

```
ffibuilder.set_source("_my_example", r"""
    #include <some_library.h>
""")
```

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

```
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 **global functions,** one for each `extern "Python"` function of the same name.

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

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:

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

and you register events by calling this function:

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

Then you would write this in the build script:

```
ffibuilder.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 *);
""")
ffibuilder.set_source("_demo_cffi", r"""
    #include <the_event_library.h>
""")
```

and in your main application you register events like this:

```
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):
        print "got event!"

@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:

```
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:

```
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):
        print "got event!"

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

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 declaration. (By default it needs to be static, but see *next paragraph*.) 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.

```
ffibuilder.set_source("_demo_cffi", r"""
    #include <the_event_library.h>

    static void my_event_callback(widget_t *, event_t *);
```

```
    /* here you can write C code which uses '&my_event_callback' */
""")
```

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):

```
ffibuilder.cdef("""
    extern "Python" int f(int);
    int my_algo(int);
""")
ffibuilder.set_source("_example_cffi", r"""
    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;
    }
""")
```

Extern "Python+C"

Functions declared with `extern "Python"` are generated as `static` functions in the C source. However, in some cases it is convenient to make them non-static, typically when you want to make them directly callable from other C source files. To do that, you can say `extern "Python+C"` instead of just `extern "Python"`. *New in version 1.6.*

if the cdef contains	then CFFI generates
`extern "Python" int f(int);`	`static int f(int) { /* code */ }`
`extern "Python+C" int f(int);`	`int f(int) { /* code */ }`

The name `extern "Python+C"` comes from the fact that we want an extern function in both senses: as an `extern "Python"`, and as a C function that is not static.

You cannot make CFFI generate additional macros or other compiler-specific stuff like the GCC `__attribute__`. You can only control whether the function should be `static` or not. But often, these attributes must be written alongside the function *header*, and it is fine if the function *implementation* does not repeat them:

```
ffibuilder.cdef("""
    extern "Python+C" int f(int);      /* not static */
""")
ffibuilder.set_source("_example_cffi", r"""
    /* the forward declaration, setting a gcc attribute
       (this line could also be in some .h file, to be included
       both here and in the other C files of the project) */
    int f(int) __attribute__((visibility("hidden")));
""")
```

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:

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

The `extern "Python"` functions cannot be variadic for now. This may be implemented in the future. (This demo shows how to do it anyway, but it is a bit lengthy.)

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 tracebacks 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']`.

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:

```
>>> @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 several issues with old-style callbacks:

- On less common architecture, libffi is more likely to crash on callbacks (e.g. on NetBSD);
- On hardened systems like PAX and SELinux, the extra memory protections can interfere (for example, on SELinux you need to run with `deny_execmem` set to `off`).

Note also that a cffi fix for the latter issue was attempted—see the `ffi_closure_alloc` branch—but was not merged because it creates potential memory corruption with `fork()`.

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:

```
# 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):
        print "method called!"
```

(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"

import cffi

ffibuilder = cffi.FFI()
ffibuilder.cdef("""
    int (*python_callback)(int how_many, int *values);
    void *const c_callback;   /* pass this const ptr to C routines */
""")
ffibuilder.set_source("_example", r"""
    #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);
    }
""")
ffibuilder.compile(verbose=True)
```

```
# file "example.py"

from _example import ffi, lib

@ffi.callback("int(int, int *)")
def python_callback(how_many, values):
    print ffi.unpack(values, how_many)
    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".*

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:

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

or:

```
ffibuilder.cdef("""
    struct foo_s {
        int (__stdcall *MyFuncPtr)(int, int);
    };
""")
```

`__cdecl` 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 mostly 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.

FFI Interface

(The reference for the FFI interface has been moved to the next page.)

CHAPTER 5

CFFI Reference

Contents

FFI Interface

This page documents the runtime interface of the two types "FFI" and "CompiledFFI". These two types are very similar to each other. You get a CompiledFFI object if you import an out-of-line module. You get a FFI object from explicitly writing cffi.FFI(). Unlike CompiledFFI, the type FFI has also got additional methods documented on the next page.

ffi.NULL

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

ffi.error

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

ffi.new()

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 the detailed documentation 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. Also, this means that a line like `x = ffi.new(...)[0]` is *always wrong:* the newly allocated object goes out of scope instantly, and so is freed immediately, and `x` is garbage.

The returned memory is initially cleared (filled with zeroes), before the optional initializer is applied. For performance, see *ffi.new_allocator()* for a way to allocate non-zero-initialized memory.

ffi.cast()

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.errno, ffi.getwinerror()

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 thread-local 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(), ffi.unpack()

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. The 'maxlen' argument limits how far we look for a null character. If 'cdata' is an array then 'maxlen' defaults to its length. See `ffi.unpack()` 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. *New in version 1.11:* can also be char16_t or char32_t.
- If 'cdata' is a single character or byte or a wchar_t or charN_t, returns it as a byte string or unicode string. (Note that in some situation a single wchar_t or char32_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.unpack(cdata, length): unpacks an array of C data of the given length, returning a Python string/unicode/list. The 'cdata' should be a pointer; if it is an array it is first converted to the pointer type. *New in version 1.6.*

- If 'cdata' is a pointer to 'char', returns a byte string. It does not stop at the first null. (An equivalent way to do that is `ffi.buffer(cdata, length)[:]`.)
- If 'cdata' is a pointer to 'wchar_t', returns a unicode string. ('length' is measured in number of wchar_t; it is not the size in bytes.) *New in version 1.11:* can also be char16_t or char32_t.
- If 'cdata' is a pointer to anything else, returns a list, of the given 'length'. (A slower way to do that is `[cdata[i] for i in range(length)]`.)

ffi.buffer(), ffi.from_buffer()

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])[:] = socket.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. (In C you might more naturally write `&array[index]`, but that is equivalent.)

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.

New in version 1.10: `ffi.buffer` is now the type of the returned buffer objects; `ffi.buffer()` actually calls the constructor.

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. It is meant to be used on objects containing large quantities of raw data, like bytearrays or `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. *In particular, never modify byte strings!*

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.

A common use case is calling a C function with some `char *` that points to the internal buffer of a Python object; for this case you can directly pass `ffi.from_buffer(python_buffer)` as argument to the call.

New in version 1.10: the `python_buffer` can be anything supporting the buffer/memoryview interface (except unicode strings). Previously, bytearray objects were supported in version 1.7 onwards (careful, if you resize the bytearray, the `<cdata>` object will point to freed memory); and byte strings were supported in version 1.8 onwards.

ffi.memmove()

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. *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`.

In versions before 1.10, `ffi.from_buffer()` had restrictions on the type of buffer, which made `ffi.memmove()` more general.

ffi.typeof(), ffi.sizeof(), ffi.alignof()

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.:

```
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.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.

For `array = ffi.new("T[]", n)`, then `ffi.sizeof(array)` returns `n * ffi.sizeof("T")`. *New in version 1.9:* Similar rules apply for structures with a variable-sized array at the end. More precisely, if `p` was returned by `ffi.new("struct foo *", ...)`, then `ffi.sizeof(p[0])` now returns the total allocated size. In previous versions, it used to just return `ffi.sizeof(ffi.typeof(p[0]))`, which is the size of the structure ignoring the variable-sized part. (Note that due to alignment, it is possible for `ffi.sizeof(p[0])` to return a value smaller than `ffi.sizeof(ffi.typeof(p[0]))`.)

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

ffi.offsetof(), ffi.addressof()

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.

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.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. 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.CData, ffi.CType

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`.

New in version 1.10: `ffi.buffer` is now *a type* as well.

ffi.gc()

ffi.gc(cdata, destructor, size=0): 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.custom_malloc(42), lib.custom_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.

ffi.gc(ptr, None, size=0): removes the ownership on a object returned by a regular call to `ffi.gc`, and no destructor will be called when it is garbage-collected. The object is modified in-place, and the function returns `None`. *New in version 1.7: ffi.gc(ptr, None)*

Note that `ffi.gc()` should be avoided for limited resources, or (with cffi below 1.11) for large memory allocations. 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.

New in version 1.11: the `size` argument. If given, this should be an estimate of the size (in bytes) that `ptr` keeps alive. This information is passed on to the garbage collector, fixing part of the problem described above. The `size` argument is most important on PyPy; on CPython, it is ignored so far, but in the future it could be used to trigger more eagerly the cyclic reference GC, too (see CPython issue 31105).

The form `ffi.gc(ptr, None, size=0)` can be called with a negative `size`, to cancel the estimate. It is not mandatory, though: nothing gets out of sync if the size estimates do not match. It only makes the next GC start more or less early.

ffi.new_handle(), ffi.from_handle()

ffi.new_handle(python_object): return a non-NULL cdata of type `void *` that contains an opaque reference to `python_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!*

See a *typical usage example* below.

(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. Usage example: suppose you

have a C library where you must call a `lib.process_document()` function which invokes some callback. The `process_document()` function receives a pointer to a callback and a `void *` argument. The callback is then invoked with the `void *data` argument that is equal to the provided value. In this typical case, you can implement it like this (out-of-line API mode):

```
class MyDocument:
    ...

    def process(self):
        h = ffi.new_handle(self)
        lib.process_document(lib.my_callback,    # the callback
                             h,                  # 'void *data'
                             args...)
        # 'h' stays alive until here, which means that the
        # ffi.from_handle() done in my_callback() during
        # the call to process_document() is safe

    def callback(self, arg1, arg2):
        ...

# the actual callback is this one-liner global function:
@ffi.def_extern()
def my_callback(arg1, arg2, data):
    return ffi.from_handle(data).callback(arg1, arg2)
```

ffi.dlopen(), ffi.dlclose()

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()

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. Example: for performance, if you are using `ffi.new()` to allocate large chunks of memory where the initial content can be left uninitialized, you can do:

```
# at module level
new_nonzero = ffi.new_allocator(should_clear_after_alloc=False)

# then replace `p = ffi.new("char[]", bigsize)` with:
    p = new_nonzero("char[]", bigsize)
```

NOTE: the following is a general warning that applies particularly (but not only) to PyPy versions 5.6 or older (PyPy > 5.6 attempts to account for the memory returned by `ffi.new()` or a custom allocator; and CPython uses reference counting). If you do large allocations, then there is no hard guarantee about when the memory will be freed. You should avoid both `new()` and `new_allocator()()` if you want to be sure that the memory is promptly released, e.g. before you allocate more of it.

An alternative is to declare and call the C `malloc()` and `free()` functions, or some variant like `mmap()` and `munmap()`. Then you control exactly when the memory is allocated and freed. For example, add these two lines to your existing `ffibuilder.cdef()`:

```
void *malloc(size_t size);
void free(void *ptr);
```

and then call these two functions manually:

```
p = lib.malloc(bigsize)
try:
    my_array = ffi.cast("some_other_type_than_void*", p)
    ...
finally:
    lib.free(p)
```

ffi.init_once()

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:

```
from _xyz_cffi import ffi, lib

def initlib():
    lib.init_my_library()

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:

```
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!")
    ...
```

ffi.getctype(), ffi.list_types()

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.list_types(): Returns the user type names known to this FFI instance. This returns a tuple containing three lists of names: `(typedef_names, names_of_structs, names_of_unions)`. *New in version 1.6.*

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.

C type	writing into	reading from	other operations
integers and enums *[5]*	an integer or anything on which int() works (but not a float!). Must be within range.	a Python int or long, depending on the type (ver. 1.10: or a bool)	int(), bool() *[6]*, <
`char`	a string of length 1 or another <cdata char>	a string of length 1	int(), bool(), <
`wchar_t`, `char16_t`, `char32_t` *[8]*	a unicode of length 1 (or maybe 2 if surrogates) or another similar <cdata>	a unicode of length 1 (or maybe 2 if surrogates)	int(), bool(), <
`float`, `double`	a float or anything on which float() works	a Python float	float(), int(), bool(), <
`long double`	another <cdata> with a `long double`, or anything on which float() works	a <cdata>, to avoid loosing precision *[3]*	float(), int(), bool()
`float _Complex`, `double _Complex`	a complex number or anything on which complex() works	a Python complex number	complex(), bool() *[7]*
pointers	another <cdata> with a compatible type (i.e. same type or `void*`, or as an array instead) *[1]*	a <cdata>	`[]` *[4]*, +, −, bool()
`void *`	another <cdata> with any pointer or array type		
pointers to structure or union	same as pointers		`[]`, +, −, bool(), and read/write struct fields
function pointers	same as pointers		bool(), call *[2]*
arrays	a list or tuple of items	a <cdata>	len(), iter(), `[]` *[4]*, +, −
`char[]`, `un/signed char[]`, `_Bool[]`	same as arrays, or a Python byte string		len(), iter(), `[]`, +, −
`wchar_t[]`, `char16_t[]`, `char32_t[]`	same as arrays, or a Python unicode string		len(), iter(), `[]`, +, −
structure	a list or tuple or dict of the field values, or a same-type <cdata>	a <cdata>	read/write fields
union	same as struct, but with at most one field		read/write fields

[1] `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, 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. (On PyPy, this optimization is only available since PyPy 5.4 with CFFI 1.8.)

[2] 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.)

[3] `long double` support:

We keep `long double` values inside a cdata object to avoid loosing 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);`.

[4] 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.)

[5] 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))`.

[6] bool() on a primitive cdata:

New in version 1.7. In previous versions, it only worked on pointers; for primitives it always returned True.

New in version 1.10: The C type `_Bool` or `bool` converts to Python booleans now. You get an exception if a C `_Bool` happens to contain a value different from 0 and 1 (this case triggers undefined behavior in C; if you really have to interface with a library relying on this, don't use `_Bool` in the CFFI side). Also, when converting from a byte string to a `_Bool[]`, only the bytes `\x00` and `\x01` are accepted.

[7] libffi does not support complex numbers:

New in version 1.11: CFFI now supports complex numbers directly. Note however that libffi does not. This means that C functions that take directly as argument types or return type a complex type cannot be called by CFFI, unless they are directly using the API mode.

[8] `wchar_t`, `char16_t` and `char32_t`

See *Unicode character types* below.

Support for 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`.

Note, however, that CFFI does this by a best-effort approach. If you need finer control over buffering, flushing, and timely closing of the `FILE *`, then you should not use this special support for `FILE *`. Instead, you can handle regular `FILE *` cdata objects that you explicitly make using fdopen(), like this:

```
ffi.cdef('''
    FILE *fdopen(int, const char *);   // from the C <stdio.h>
    int fclose(FILE *);
''')

myfile.flush()                      # make sure the file is flushed
newfd = os.dup(myfile.fileno())     # make a copy of the file descriptor
fp = lib.fdopen(newfd, "w")         # make a cdata 'FILE *' around newfd
lib.write_stuff_to_file(fp)         # invoke the external function
lib.fclose(fp)                      # when you're done, close fp (and newfd)
```

The special support for `FILE *` is anyway implemented in a similar manner on CPython 3.x and on PyPy, because these Python implementations' files are not natively based on `FILE *`. Doing it explicity offers more control.

Unicode character types

The `wchar_t` type has the same signedness as the underlying platform's. For example, on Linux, it is a signed 32-bit integer. However, the types `char16_t` and `char32_t` (*new in version 1.11*) are always unsigned.

Note that CFFI assumes that these types are meant to contain UTF-16 or UTF-32 characters in the native endianness. More precisely:

- `char32_t` is assumed to contain UTF-32, or UCS4, which is just the unicode codepoint;
- `char16_t` is assumed to contain UTF-16, i.e. UCS2 plus surrogates;
- `wchar_t` is assumed to contain either UTF-32 or UTF-16 based on its actual platform-defined size of 4 or 2 bytes.

Whether this assumption is true or not is unspecified by the C language. In theory, the C library you are interfacing with could use one of these types with a different meaning. You would then need to handle it yourself—for example, by using `uint32_t` instead of `char32_t` in the `cdef()`, and building the expected arrays of `uint32_t` manually.

Python itself can be compiled with `sys.maxunicode == 65535` or `sys.maxunicode == 1114111` (Python >= 3.3 is always 1114111). This changes the handling of surrogates (which are pairs of 16-bit "characters" which actually stand for a single codepoint whose value is greater than 65535). If your Python is `sys.maxunicode == 1114111`, then it can store arbitrary unicode codepoints; surrogates are automatically inserted when converting from Python unicodes to UTF-16, and automatically removed when converting back. On the other hand, if your Python is `sys.maxunicode == 65535`, then it is the other way around: surrogates are removed when converting from Python unicodes to UTF-32, and added when converting back. In other words, surrogate conversion is done only when there is a size mismatch.

Note that Python's internal representations is not specified. For example, on CPython >= 3.3, it will use 1- or 2- or 4-bytes arrays depending on what the string actually contains. With CFFI, when you pass a Python byte string to a C function expecting a `char*`, then we pass directly a pointer to the existing data without needing a temporary buffer; however, the same cannot cleanly be done with *unicode* string arguments and the `wchar_t*` / `char16_t*` / `char32_t*` types, because of the changing internal representation. As a result, and for consistency, CFFI always allocates a temporary buffer for unicode strings.

Warning: for now, if you use `char16_t` and `char32_t` with `set_source()`, you have to make sure yourself that the types are declared by the C source you provide to `set_source()`. They would be declared if you `#include` a library that explicitly uses them, for example, or when using C++11. Otherwise, you need `#include <uchar.h>` on Linux, or more generally something like `typedef uint16_t char16_t;`. This is not done automatically by CFFI because `uchar.h` is not standard across platforms, and writing a `typedef` like above would crash if the type happens to be already defined.

CHAPTER 6

Preparing and Distributing modules

Contents

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

- The **"in-line", "ABI mode"**:

```
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:

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

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

if __name__ == "__main__":
    ffibuilder.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 and speed 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 cffi

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

if __name__ == "__main__":
    ffibuilder.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 CFFI 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.ffibuilder.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:ffibuilder"],
    install_requires=["cffi>=1.0.0"],
)
```

- Note that some bundler tools that try to find all modules used by a project, like PyInstaller, will miss `_cffi_backend` in the out-of-line mode because your program contains no explicit `import cffi` or `import _cffi_backend`. You need to add `_cffi_backend` explicitly (as a "hidden import" in PyInstaller, but it can also be done more generally by adding the line `import _cffi_backend` in your main program).

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_or_ffibuilder = cffi.FFI()`. It has the same functionality (for in-line use), but also the extra methods described below (to prepare the FFI). NOTE: We use the name `ffibuilder` instead of `ffi` in the out-of-line context, when the code is about producing a `_foo.so` file; this is an attempt to distinguish it from the different `ffi` object that you get by later saying `from _foo import 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; 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__`, `lib.__all__` and `lib.__name__` added in successive versions.

ffi/ffibuilder.cdef(): declaring types and functions

ffi/ffibuilder.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, ptrdiff_t, size_t, ssize_t
- wchar_t (if supported by the backend). *New in version 1.11:* char16_t and char32_t.
- _Bool and bool (equivalent). If not directly supported by the C compiler, this is declared with the size of `unsigned char`.
- FILE. See here.

- all common Windows types are defined if you run on Windows (`DWORD`, `LPARAM`, etc.). Exception: `TBYTE TCHAR LPCTSTR PCTSTR LPTSTR PTSTR PTBYTE PTCHAR` are not automatically defined; see *ffi.set_unicode()*.
- 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:

```
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
    );
  """))
```

Note also that pycparser, the underlying C parser, recognizes preprocessor-like directives in the following format: `# NUMBER "FILE"`. For example, if you put `# 42 "foo.h"` in the middle of the string passed to `cdef()` and there is an error two lines later, then it is reported with an error message that starts with `foo.h:43:` (the line which is given the number 42 is the line immediately after the directive). *New in version 1.10.1:* CFFI automatically puts the line `# 1 "<cdef source string>"` just before the string you give to `cdef()`.
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()`.)

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.

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 (dependency on ABI details, getting crashes instead of C compiler errors/warnings, and higher overhead to call the C functions). 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). Note that `libpath` cannot be None on Windows with Python 3.

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. API-level access, on the other hand, is safer. Speed-wise, API-level access is much faster (it is common to have the opposite misconception about performance).

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'))`.

ffibuilder.set_source(): preparing out-of-line modules

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

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

In **ABI mode,** you call `ffibuilder.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. Typically, it is specified as `r""" ...multiple lines of C code... """` (the `r` prefix allows these lines to contain a literal `\n`, for example). 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 file "module_name.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):

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

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 `cdef()` as taking a `int`, but actually takes a `long`, then the C compiler handles the difference.
- other arguments are checked: you get a compilation warning or error if you pass a `int *` argument to a function expecting a `long *`.
- similarly, most other things declared in the `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 `cdef()` at various places, in order to ask the C compiler to fill in the details. These places are:

- structure declarations: any `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 `struct` declaration which doesn't use "`...`" is assumed to be exact, but this is checked: you get an error if it is not correct.
- integer types: the syntax "`typedef int... foo_t;`" declares the type `foo_t` as an integer type whose exact size and signedness is not specified. The compiler will figure it out. (Note that this requires `set_source()`; it does not work with `verify()`.) The `int...` can be replaced with `long...` or `unsigned long long...` or any other primitive integer type, with no effect. The type will always map to one of `(u)int(8,16,32,64)_t` in Python, but in the generated C code, only `foo_t` is used.

- *New in version 1.3:* floating-point types: "`typedef float... foo_t;`" (or equivalently "`typedef double... foo_t;`") declares `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 (`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 `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 "`typedef ... foo_t;`" declares the type `foo_t` as opaque. Useful mainly for when the API takes and returns `foo_t *` without you needing to look inside the `foo_t`. Also works with "`typedef ... *foo_p;`" which declares the pointer type `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). *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 `foo_t` is not opaque, but just a struct where you don't know any field; then you would use "`typedef struct { ...; } foo_t;`".

- array lengths: when used as structure fields or in global variables, arrays can have an unspecified length, as in "`int n[...];`". The length is completed by the C compiler. This is slightly different from "`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. This supports multidimensional arrays: "`int n[...][...];`".

  New in version 1.2: "`int m[][...];`", i.e. `...` can be used in the innermost dimensions without being also used in the outermost dimension. In the example given, the length of the `m` array is assumed not to be known to the C compiler, but the length of every item (like the sub-array `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 `[...]` in CFFI.

- enums: if you don't know the exact order (or values) of the declared constants, then use this syntax: "`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 "`enum foo { A=..., B, C };`" or even "`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 `cdef` the line "`#define FOO ...`", with any macro name FOO but with `...` 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 `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 `const` together with the variable name, as in `static char *const FOO;`).

Currently, it is not supported to find automatically which of the various integer or float types you need at which place—except in the following case: if such a type is explicitly named. For an integer type, use `typedef int... the_type_name;`, or another type like `typedef unsigned long... the_type_name;`. Both are equivalent and replaced by the real C type, which must be an integer type. Similarly, for floating-point types, use `typedef float... the_type_name;` or equivalently `typedef double... the_type_name;`. Note that `long double` cannot be detected this way.

In the case of function arguments or return types, when it is a simple integer/float type, you can simply misdeclare it. If you misdeclare a function `void f(long)` as `void f(int)`, it still works (but you have to call it with arguments that fit an int). It works because the C compiler will do the casting for us. This C-level casting of arguments and return types only works for regular function, and not for function pointer types; currently, it also does not work for variadic functions.

For more complex types, you have no choice but be precise. For example, you cannot misdeclare a `int *` argument as `long *`, or a global array `int a[5];` as `long a[5];`. CFFI considers *all types listed above* as primitive (so `long long a[5];` and `int64_t a[5]` are different declarations). The reason for that is detailed in a comment about an issue.

ffibuilder.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 `ffibuilder.set_source()` and `ffibuilder.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.

New in version 1.8: the C code produced by `emit_c_code()` or `compile()` contains `#define Py_LIMITED_API`. This means that on CPython >= 3.2, compiling this source produces a binary .so/.dll that should work for any version of CPython >= 3.2 (as opposed to only for the same version of CPython x.y). However, the standard `distutils` package will still produce a file called e.g. `NAME.cpython-35m-x86_64-linux-gnu.so`. You can manually rename it to `NAME.abi3.so`, or use setuptools version 26 or later. Also, note that compiling with a debug version of Python will not actually define `Py_LIMITED_API`, as doing so makes `Python.h` unhappy.

ffibuilder.compile(tmpdir='.', verbose=False, debug=None): 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 __name__ == "__main__": ffibuilder.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.)

New in version 1.8.1: `debug` argument. If set to a bool, it controls whether the C code is compiled in debug mode or not. The default None means to use the host Python's `sys.flags.debug`. Starting with version 1.8.1, if you are running a debug-mode Python, the C code is thus compiled in debug mode by default (note that it is anyway necessary to do so on Windows).

ffibuilder.emit_python_code(filename): generate the given .py file (same as `ffibuilder.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).

ffibuilder.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`).

ffibuilder.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 `cffi_modules=["path/to/foo_build.py:ffibuilder"]` 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 `execfile()`) and looks up its global variable called `ffibuilder`. You can also say `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.

ffi/ffibuilder.include(): combining multiple CFFI interfaces

ffi/ffibuilder.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 `ffibuilder.include(other_ffibuilder)` line should occur in the build script, and the `other_ffibuilder` argument should be another FFI instance 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`.

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 either in C99, or are GCC or MSVC extensions:

- Any `__attribute__` or `#pragma pack(n)`
- Additional types: special-size floating and fixed point types, vector types, and so on.
- The C99 types `float _Complex` and `double _Complex` are supported by cffi since version 1.11, but not libffi: you cannot call C functions with complex arguments or return value, except if they are directly API-mode functions. The type `long double _Complex` is not supported at all (declare and use it as if it were an array of two `long double`, and write wrapper functions in C with set_source()).
- **`__restrict__` or `__restrict` are extensions of, respectively,** GCC and MSVC. They are not recognized. But `restrict` is a C keyword and is accepted (and ignored).

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.

Note that if you declare a variable in `cdef()` without using `const`, CFFI assumes it is a read-write variable and generates two pieces of code, one to read it and one to write it. If the variable cannot in fact be written to in C code, for one reason or another, it will not compile. In this case, you can declare it as a constant: for example, instead of `foo_t *myglob;` you would use `foo_t *const myglob;`. Note also that `const foo_t *myglob;` is a *variable;* it contains a variable pointer to a constant `foo_t`.

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 intented 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.

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):

```
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>`. Useful 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 `ffibuilder.set_source()`.
- The optional `flags` argument (ignored on Windows) defaults to `ffi.RTLD_NOW`; see `man dlopen`. (With `ffibuilder.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:

```
ext = ffi.verify(..., sources=['foo.c'], relative_to=__file__)
```

The line above is roughly the same as:

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

except that the default name of the produced library is built from the CRC checkum 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 `cffi.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`.

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()`:

```
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()`:

```
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 old versions of PyPy (CFFI 1.0 does not work in PyPy < 2.6):

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

ffi = cffi.FFI()
C_HEADER_SRC = r'''
```

```
    #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_package="...",     # 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
)
```

CHAPTER 7

Using CFFI for embedding

Contents

You can use CFFI to generate C code which exports the API of your choice to any C application that wants to link with this C code. This API, which you define yourself, ends up as the API of a `.so`/`.dll`/`.dylib` library—or you can statically link it within a larger application.

Possible use cases:

- Exposing a library written in Python directly to C/C++ programs.
- Using Python to make a "plug-in" for an existing C/C++ program that is already written to load them.
- Using Python to implement part of a larger C/C++ application (with static linking).
- Writing a small C/C++ wrapper around Python, hiding the fact that the application is actually written in Python (to make a custom command-line interface; for distribution purposes; or simply to make it a bit harder to reverse-engineer the application).

The general idea is as follows:

- You write and execute a Python script, which produces a `.c` file with the API of your choice (and optionally compile it into a `.so/.dll/.dylib`). The script also gives some Python code to be "frozen" inside the `.so`.
- At runtime, the C application loads this `.so/.dll/.dylib` (or is statically linked with the `.c` source) without having to know that it was produced from Python and CFFI.
- The first time a C function is called, Python is initialized and the frozen Python code is executed.
- The frozen Python code defines more Python functions that implement the C functions of your API, which are then used for all subsequent C function calls.

One of the goals of this approach is to be entirely independent from the CPython C API: no `Py_Initialize()` nor `PyRun_SimpleString()` nor even `PyObject`. It works identically on CPython and PyPy.

This is entirely *new in version 1.5.* (PyPy contains CFFI 1.5 since release 5.0.)

Usage

See the paragraph in the overview page for a quick introduction. In this section, we explain every step in more details. We will use here this slightly expanded example:

```
/* file plugin.h */
typedef struct { int x, y; } point_t;
extern int do_stuff(point_t *);
```

```
/* file plugin.h, Windows-friendly version */
typedef struct { int x, y; } point_t;

/* When including this file from ffibuilder.set_source(), the
   following macro is defined to '__declspec(dllexport)'.  When
   including this file directly from your C program, we define
   it to 'extern __declspec(dllimport)' instead.

   With non-MSVC compilers we simply define it to 'extern'.
   (The 'extern' is needed for sharing global variables;
   functions would be fine without it.  The macros always
   include 'extern': you must not repeat it when using the
   macros later.)
*/
#ifndef CFFI_DLLEXPORT
#  if defined(_MSC_VER)
#    define CFFI_DLLEXPORT  extern __declspec(dllimport)
#  else
#    define CFFI_DLLEXPORT  extern
#  endif
#endif

CFFI_DLLEXPORT int do_stuff(point_t *);
```

```
# file plugin_build.py
import cffi
ffibuilder = cffi.FFI()

with open('plugin.h') as f:
    # read plugin.h and pass it to embedding_api(), manually
    # removing the '#' directives and the CFFI_DLLEXPORT
    data = ''.join([line for line in f if not line.startswith('#')])
```

```
    data = data.replace('CFFI_DLLEXPORT', '')
    ffibuilder.embedding_api(data)

ffibuilder.set_source("my_plugin", r'''
    #include "plugin.h"
''')

ffibuilder.embedding_init_code("""
    from my_plugin import ffi

    @ffi.def_extern()
    def do_stuff(p):
        print("adding %d and %d" % (p.x, p.y))
        return p.x + p.y
""")

ffibuilder.compile(target="plugin-1.5.*", verbose=True)
# or: ffibuilder.emit_c_code("my_plugin.c")
```

Running the code above produces a *DLL*, i,e, a dynamically-loadable library. It is a file with the extension `.dll` on Windows, `.dylib` on Mac OS/X, or `.so` on other platforms. As usual, it is produced by generating some intermediate `.c` code and then calling the regular platform-specific C compiler. See *below* for some pointers to C-level issues with using the produced library. Here are some details about the methods used above:

- **ffibuilder.embedding_api(source):** parses the given C source, which declares functions that you want to be exported by the DLL. It can also declare types, constants and global variables that are part of the C-level API of your DLL.

  The functions that are found in `source` will be automatically defined in the `.c` file: they will contain code that initializes the Python interpreter the first time any of them is called, followed by code to call the attached Python function (with `@ffi.def_extern()`, see next point).

  The global variables, on the other hand, are not automatically produced. You have to write their definition explicitly in `ffibuilder.set_source()`, as regular C code (see the point after next).

- **ffibuilder.embedding_init_code(python_code):** this gives initialization-time Python source code. This code is copied ("frozen") inside the DLL. At runtime, the code is executed when the DLL is first initialized, just after Python itself is initialized. This newly initialized Python interpreter has got an extra "built-in" module that can be loaded magically without accessing any files, with a line like "`from my_plugin import ffi, lib`". The name `my_plugin` comes from the first argument to `ffibuilder.set_source()`. This module represents "the caller's C world" from the point of view of Python.

  The initialization-time Python code can import other modules or packages as usual. You may have typical Python issues like needing to set up `sys.path` somehow manually first.

  For every function declared within `ffibuilder.embedding_api()`, the initialization-time Python code or one of the modules it imports should use the decorator `@ffi.def_extern()` to attach a corresponding Python function to it.

  If the initialization-time Python code fails with an exception, then you get a traceback printed to stderr, along with more information to help you identify problems like wrong `sys.path`. If some function remains unattached at the time where the C code tries to call it, an error message is also printed to stderr and the function returns zero/null.

  Note that the CFFI module never calls `exit()`, but CPython itself contains code that calls `exit()`, for example if importing `site` fails. This may be worked around in the future.

- **ffibuilder.set_source(c_module_name, c_code):** set the name of the module from Python's point of view. It also gives more C code which will be included in the generated C code. In trivial examples it can be an

empty string. It is where you would `#include` some other files, define global variables, and so on. The macro `CFFI_DLLEXPORT` is available to this C code: it expands to the platform-specific way of saying "the following declaration should be exported from the DLL". For example, you would put "`extern int my_glob;`" in `ffibuilder.embedding_api()` and "`CFFI_DLLEXPORT int my_glob = 42;`" in `ffibuilder.set_source()`.

Currently, any *type* declared in `ffibuilder.embedding_api()` must also be present in the `c_code`. This is automatic if this code contains a line like `#include "plugin.h"` in the example above.

- **ffibuilder.compile([target=...] [, verbose=True]):** make the C code and compile it. By default, it produces a file called `c_module_name.dll`, `c_module_name.dylib` or `c_module_name.so`, but the default can be changed with the optional `target` keyword argument. You can use `target="foo.*"` with a literal `*` to ask for a file called `foo.dll` on Windows, `foo.dylib` on OS/X and `foo.so` elsewhere. One reason for specifying an alternate `target` is to include characters not usually allowed in Python module names, like "`plugin-1.5.*`".

  For more complicated cases, you can call instead `ffibuilder.emit_c_code("foo.c")` and compile the resulting `foo.c` file using other means. CFFI's compilation logic is based on the standard library `distutils` package, which is really developed and tested for the purpose of making CPython extension modules; it might not always be appropriate for making general DLLs. Also, just getting the C code is what you need if you do not want to make a stand-alone `.so/.dll/.dylib` file: this C file can be compiled and statically linked as part of a larger application.

More reading

If you're reading this page about embedding and you are not familiar with CFFI already, here are a few pointers to what you could read next:

- For the `@ffi.def_extern()` functions, integer C types are passed simply as Python integers; and simple pointers-to-struct and basic arrays are all straightforward enough. However, sooner or later you will need to read about this topic in more details here.
- `@ffi.def_extern()`: see documentation here, notably on what happens if the Python function raises an exception.
- To create Python objects attached to C data, one common solution is to use `ffi.new_handle()`. See documentation here.
- In embedding mode, the major direction is C code that calls Python functions. This is the opposite of the regular extending mode of CFFI, in which the major direction is Python code calling C. That's why the page Using the ffi/lib objects talks first about the latter, and why the direction "C code that calls Python" is generally referred to as "callbacks" in that page. If you also need to have your Python code call C code, read more about *Embedding and Extending* below.
- `ffibuilder.embedding_api(source)`: follows the same syntax as `ffibuilder.cdef()`, documented here. You can use the "`...`" syntax as well, although in practice it may be less useful than it is for `cdef()`. On the other hand, it is expected that often the C sources that you need to give to `ffibuilder.embedding_api()` would be exactly the same as the content of some `.h` file that you want to give to users of your DLL. That's why the example above does this:

```
with open('foo.h') as f:
    ffibuilder.embedding_api(f.read())
```

  Note that a drawback of this approach is that `ffibuilder.embedding_api()` doesn't support `#ifdef` directives. You may have to use a more convoluted expression like:

```
with open('foo.h') as f:
    lines = [line for line in f if not line.startswith('#')]
    ffibuilder.embedding_api(''.join(lines))
```

As in the example above, you can also use the same `foo.h` from `ffibuilder.set_source()`:

```
ffibuilder.set_source('module_name', r'''
    #include "foo.h"
''')
```

Troubleshooting

The error message

> cffi extension module 'c_module_name' has unknown version 0x2701

means that the running Python interpreter located a CFFI version older than 1.5. CFFI 1.5 or newer must be installed in the running Python.

Issues about using the .so

This paragraph describes issues that are not necessarily specific to CFFI. It assumes that you have obtained the `.so`/`.dylib`/`.dll` file as described above, but that you have troubles using it. (In summary: it is a mess. This is my own experience, slowly built by using Google and by listening to reports from various platforms. Please report any inaccuracies in this paragraph or better ways to do things.)

- The file produced by CFFI should follow this naming pattern: `libmy_plugin.so` on Linux, `libmy_plugin.dylib` on Mac, or `my_plugin.dll` on Windows (no `lib` prefix on Windows).
- First note that this file does not contain the Python interpreter nor the standard library of Python. You still need it to be somewhere. There are ways to compact it to a smaller number of files, but this is outside the scope of CFFI (please report if you used some of these ways successfully so that I can add some links here).
- In what we'll call the "main program", the `.so` can be either used dynamically (e.g. by calling `dlopen()` or `LoadLibrary()` inside the main program), or at compile-time (e.g. by compiling it with `gcc -lmy_plugin`). The former case is always used if you're building a plugin for a program, and the program itself doesn't need to be recompiled. The latter case is for making a CFFI library that is more tightly integrated inside the main program.
- In the case of compile-time usage: you can add the gcc option `-Lsome/path/` before `-lmy_plugin` to describe where the `libmy_plugin.so` is. On some platforms, notably Linux, `gcc` will complain if it can find `libmy_plugin.so` but not `libpython27.so` or `libpypy-c.so`. To fix it, you need to call `LD_LIBRARY_PATH=/some/path/to/libpypy gcc`.
- When actually executing the main program, it needs to find the `libmy_plugin.so` but also `libpython27.so` or `libpypy-c.so`. For PyPy, unpack a PyPy distribution and you get a full directory structure with `libpypy-c.so` inside a `bin` subdirectory, or on Windows `pypy-c.dll` inside the top directory; you must not move this file around, but just point to it. One way to point to it is by running the main program with some environment variable: `LD_LIBRARY_PATH=/some/path/to/libpypy` on Linux, `DYLD_LIBRARY_PATH=/some/path/to/libpypy` on OS/X.
- You can avoid the `LD_LIBRARY_PATH` issue if you compile `libmy_plugin.so` with the path hard-coded inside in the first place. On Linux, this is done by `gcc -Wl,-rpath=/some/path`. You would

put this option in `ffibuilder.set_source("my_plugin", ..., extra_link_args=['-Wl,-rpath=/some/path/to/libpypy'])`. The path can start with `$ORIGIN` to mean "the directory where `libmy_plugin.so` is". You can then specify a path relative to that place, like `extra_link_args=['-Wl,-rpath=$ORIGIN/../venv/bin']`. Use `ldd libmy_plugin.so` to look at what path is currently compiled in after the expansion of `$ORIGIN`.)

After this, you don't need `LD_LIBRARY_PATH` any more to locate `libpython27.so` or `libpypy-c.so` at runtime. In theory it should also cover the call to `gcc` for the main program. I wasn't able to make `gcc` happy without `LD_LIBRARY_PATH` on Linux if the rpath starts with `$ORIGIN`, though.

- The same rpath trick might be used to let the main program find `libmy_plugin.so` in the first place without `LD_LIBRARY_PATH`. (This doesn't apply if the main program uses `dlopen()` to load it as a dynamic plugin.) You'd make the main program with `gcc -Wl,-rpath=/path/to/libmyplugin`, possibly with `$ORIGIN`. The `$` in `$ORIGIN` causes various shell problems on its own: if using a common shell you need to say `gcc -Wl,-rpath=\$ORIGIN`. From a Makefile, you need to say something like `gcc -Wl,-rpath=\$$ORIGIN`.
- On some Linux distributions, notably Debian, the `.so` files of CPython C extension modules may be compiled without saying that they depend on `libpythonX.Y.so`. This makes such Python systems unsuitable for embedding if the embedder uses `dlopen(..., RTLD_LOCAL)`. You get the error `undefined symbol: PyExc_SystemError`. See issue #264.

Using multiple CFFI-made DLLs

Multiple CFFI-made DLLs can be used by the same process.

Note that all CFFI-made DLLs in a process share a single Python interpreter. The effect is the same as the one you get by trying to build a large Python application by assembling a lot of unrelated packages. Some of these might be libraries that monkey-patch some functions from the standard library, for example, which might be unexpected from other parts.

Multithreading

Multithreading should work transparently, based on Python's standard Global Interpreter Lock.

If two threads both try to call a C function when Python is not yet initialized, then locking occurs. One thread proceeds with initialization and blocks the other thread. The other thread will be allowed to continue only when the execution of the initialization-time Python code is done.

If the two threads call two *different* CFFI-made DLLs, the Python initialization itself will still be serialized, but the two pieces of initialization-time Python code will not. The idea is that there is a priori no reason for one DLL to wait for initialization of the other DLL to be complete.

After initialization, Python's standard Global Interpreter Lock kicks in. The end result is that when one CPU progresses on executing Python code, no other CPU can progress on executing more Python code from another thread of the same process. At regular intervals, the lock switches to a different thread, so that no single thread should appear to block indefinitely.

Testing

For testing purposes, a CFFI-made DLL can be imported in a running Python interpreter instead of being loaded like a C shared library.

You might have some issues with the file name: for example, on Windows, Python expects the file to be called `c_module_name.pyd`, but the CFFI-made DLL is called `target.dll` instead. The base name `target` is the one specified in `ffibuilder.compile()`, and on Windows the extension is `.dll` instead of `.pyd`. You have to rename or copy the file, or on POSIX use a symlink.

The module then works like a regular CFFI extension module. It is imported with "`from c_module_name import ffi, lib`" and exposes on the `lib` object all C functions. You can test it by calling these C functions. The initialization-time Python code frozen inside the DLL is executed the first time such a call is done.

Embedding and Extending

The embedding mode is not incompatible with the non-embedding mode of CFFI.

You can use *both* `ffibuilder.embedding_api()` and `ffibuilder.cdef()` in the same build script. You put in the former the declarations you want to be exported by the DLL; you put in the latter only the C functions and types that you want to share between C and Python, but not export from the DLL.

As an example of that, consider the case where you would like to have a DLL-exported C function written in C directly, maybe to handle some cases before calling Python functions. To do that, you must *not* put the function's signature in `ffibuilder.embedding_api()`. (Note that this requires more hacks if you use `ffibuilder.embedding_api(f.read())`.) You must only write the custom function definition in `ffibuilder.set_source()`, and prefix it with the macro CFFI_DLLEXPORT:

```
CFFI_DLLEXPORT int myfunc(int a, int b)
{
    /* implementation here */
}
```

This function can, if it wants, invoke Python functions using the general mechanism of "callbacks"—called this way because it is a call from C to Python, although in this case it is not calling anything back:

```
ffibuilder.cdef("""
    extern "Python" int mycb(int);
""")

ffibuilder.set_source("my_plugin", r"""

    static int mycb(int);   /* the callback: forward declaration, to make
                               it accessible from the C code that follows */

    CFFI_DLLEXPORT int myfunc(int a, int b)
    {
        int product = a * b;   /* some custom C code */
        return mycb(product);
    }
""")
```

and then the Python initialization code needs to contain the lines:

```
@ffi.def_extern()
def mycb(x):
    print "hi, I'm called with x =", x
    return x * 10
```

This `@ffi.def_extern` is attaching a Python function to the C callback `mycb()`, which in this case is not exported from the DLL. Nevertheless, the automatic initialization of Python occurs when `mycb()` is called, if it happens to

be the first function called from C. More precisely, it does not happen when `myfunc()` is called: this is just a C function, with no extra code magically inserted around it. It only happens when `myfunc()` calls `mycb()`.

As the above explanation hints, this is how `ffibuilder.embedding_api()` actually implements function calls that directly invoke Python code; here, we have merely decomposed it explicitly, in order to add some custom C code in the middle.

In case you need to force, from C code, Python to be initialized before the first `@ffi.def_extern()` is called, you can do so by calling the C function `cffi_start_python()` with no argument. It returns an integer, 0 or -1, to tell if the initialization succeeded or not. Currently there is no way to prevent a failing initialization from also dumping a traceback and more information to stderr.

CHAPTER 8

Goals

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.

CHAPTER 9

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