Python CFFI simple tutorial

Notify:

Cuong Manh Do, Toan Xuan, Mai Unknown User (danght), Unknown User (kimtn), Unknown User (linhht), Unknown User (hanhtkt), Unknown User (quidvt.jd), Unknown User (tandn.jd)

Ref:

http://cffi.readthedocs.org/

Overview

CFFI can be used in one of four modes: “ABI” versus “API” level, each with “in-line” or “out-of-line” preparation (or compilation).

The **ABI mode** accesses libraries at the binary level, whereas the **API mode** accesses them with a C compiler. This is described in detail below.

In the **in-line mode,** everything is set up every time you import your Python code. In the **out-of-line mode,** you have a separate step of preparation (and possibly C compilation) that produces a module which your main program can then import.

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

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)`.

This example does not call any C compiler.

Out-of-line example (ABI level, out-of-line)

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

This example does not call any C compiler.

```
# file "simple_example_build.py"

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

from cffi import FFI

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

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

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

```
from _simple_example import ffi

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

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

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

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

```
from setuptools import setup

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

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

```
# file "example_build.py"

from cffi import FFI
ffi = FFI()

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

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

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

You need to run the `example_build.py` script once to generate “source code” into the file `_example.c` and compile this to a regular C extension module. (CFFI selects either Python or C for the module to generate based on whether the second argument to `set_source()` is `Non e` 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.

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

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
ffi = FFI()

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

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

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

```
# 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.

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. You are not meant to access fields by guessing where they are in the structures. *The C libraries are typically meant to be used with a C compiler.*

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

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

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

The generated piece of C code should be the same independently on the platform on which you run it (or the Python version), so in simple cases you can directly distribute the pre-generated C code and treat it as a regular C extension module. The special Setuptools lines in the example above are meant for the more complicated cases where we need to regenerate the C sources as well—e.g. because the Python script that regenerates this file will itself look around the system to know what it should include or not.

Note that the "API level + in-line" mode combination is deprecated. It used to be done with `lib = ffi.verify("C header")`. The out-of-line variant with `set_source("modname", "C header")` is preferred.

Using the ffi/lib objects

Working with pointers, structures and arrays

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

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

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

`ffi.new(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
```

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). But see ffi.addressof().

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

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

Example:

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

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

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

```
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` nor mally 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 *` arguments. Note that in general, there is no difference between C argument declarations that use `type *` or `type[]`. For example, `int *` is fully equivalent to `int[]` (or even `int[5]`; the 5 is ignored). So you can pass an `int *` as a list of integers:
```
# void do_something_with_array(int *array);

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

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

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

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

Aside from these limitations, functions and callbacks can receive and return structs.

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

Before version 1.1, if you really need a cdata pointer to the function, use the following workaround:

```
ffi.cdef(""" int (*foo)(int a, int b); """)
```

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

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();` rea lly means `int foo(void);`.)

Callbacks

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: 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.

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

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

```
# file "example.py"

from _example import ffi, lib

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

Be careful when writing the Python callback function: if it returns an object of the wrong type, or more generally raises an exception, then the exception cannot be propagated. Instead, it is printed to stderr and the C-level callback is made to return a default value.

The returned value in case of errors is 0 or null by default, but can be specified with the `error` keyword argument to `ffi.callback()`:
`@ffi.callback("int(int, int)", error=-1)`

The exception is still printed to stderr, so this should be used only as a last-resort solution.

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.
Warning

SELinux requires that the setting `deny_execmem` is left to its default setting of `off` to use callbacks. A fix in cffi was attempted (see the `ffi_closure_alloc` branch), but this branch is not merged because it creates potential memory corruption with `fork()`. For more information, see here.

New in version 1.2: If you want to be sure to catch all exceptions, use `ffi.callback(..., onerror=func)`. 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, it will be called after entering the main callback function but before executing the `try:`.

If `onerror` 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 one invoked by the callback handler. So you can get the value of `argname` in that frame by reading `traceback.tb_frame.f_locals['argname']`.

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:

```
ffi.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 not needed (but doesn't hurt) to say `WINAPI` or `__stdcall` when declaring a plain function in the `cdef()`.

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

ffi.new(cdecl, init=None): allocate an instance according to the specified C type and return a pointer to it. The specified C type must be either a pointer or an array: `new('X *')` allocates an X and returns a pointer to it, whereas `new('X[n]')` allocates an array of n X'es and returns an array referencing it (which works mostly like a pointer, like in C). You can also use `new('X[]', n)` to allocate an array of a non-constant length n. See above for other valid initializers.

When the returned `<cdata>` object goes out of scope, the memory is freed. In other words the returned `<cdata>` object has ownership of the value of type `cdecl` that it points to. This means that the raw data can be used as long as this object is kept alive, but must not be used for a longer time. Be careful about that when copying the pointer to the memory somewhere else, e.g. into another structure.

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

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

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

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

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

- If 'cdata' is a pointer or array of characters or bytes, returns the null-terminated string. The returned string extends until the first null character, or at most 'maxlen' characters. If 'cdata' is an array then 'maxlen' defaults to its length. See `ffi.buffer()` below for a way to continue past the first null character. *Python 3:* this returns a `bytes`, not a `str`.
- If 'cdata' is a pointer or array of wchar_t, returns a unicode string following the same rules.
- If 'cdata' is a single character or byte or a wchar_t, returns it as a byte string or unicode string. (Note that in some situation a single wchar_t may require a Python unicode string of length 2.)
- If 'cdata' is an enum, returns the value of the enumerator as a string. If the value is out of range, it is simply returned as the stringified integer.

ffi.buffer(cdata, [size]): return a buffer object that references the raw C data pointed to by the given 'cdata', of 'size' bytes. The 'cdata' must be a pointer or an array. If unspecified, the size of the buffer is either the size of what `cdata` points to, or the whole size of the array. Getting a buffer is useful because you can read from it without an extra copy, or write into it to change the original value.

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

- use `file.write()` and `file.readinto()` with such a buffer (for files opened in binary mode)
- use `ffi.buffer(mystruct[0])[:] = 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.

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

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

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

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

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

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

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

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

```
def myfunction(ptr):
    assert ffi.typeof(ptr) is ffi.typeof("foo_t*")
    ...
```

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

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

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

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

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

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

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

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

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

Note that this should be avoided for large memory allocations or for limited resources. This is particularly true on PyPy: its GC does not know how much memory or how many resources the returned `ptr` holds. It will only run its GC when enough memory it knows about has been allocated (and thus run the destructor possibly later than you would expect). Moreover, the destructor is called in whatever thread PyPy is at that moment, which might be a problem for some C libraries. In these cases, consider writing a wrapper class with custom `__enter__()` and `__exit__()` methods, allocating and freeing the C data at known points in time, and using it in a `with` statement.

ffi.new_handle(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 retrive the original `python_object` from a value with the same `void *` pointer. *Calling ffi.from_handle(p) is invalid and will likely crash if the cdata object returned by new_handle() is not kept alive!*

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

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

- `new_handle()` returns a cdata object that contains a reference to the Python object; we call them collectively the "handle" cdata objects. The `void *` value in this handle cdata object is 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.

ffi.addressof(cdata, *fields_or_indexes): limited equivalent to the '&' operator in C:

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

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

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

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

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

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

ffi.RLTD_...: constants: flags for `ffi.dlopen()`.

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

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

If `should_clear_after_alloc` is set to False, then the memory returned by `alloc()` is assumed to be already cleared (or you are fine with garbage); otherwise CFFI will clear it.

Reference: conversions

This section documents all the conversions that are allowed when *writing into* a C data structure (or passing arguments to a function call), and *reading from* a C data structure (or getting the result of a function call). The last column gives the type-specific operations allowed.

C type	writing into	reading from	other operations
integers and enums —(*****)	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	int()
`char`	a string of length 1 or another <cdata char>	a string of length 1	int()
`wchar_t`	a unicode of length 1 (or maybe 2 if surrogates) or another <cdata wchar_t>	a unicode of length 1 (or maybe 2 if surrogates)	int()
`float`, `double`	a float or anything on which float() works	a Python float	float(), int()
`long double`	another <cdata> with a `long double`, or anything on which float() works	a <cdata>, to avoid loosing precision —(***)	float(), int()
pointers	another <cdata> with a compatible type (i.e. same type or `char*` or `void*`, or as an array instead) —(*)	a <cdata>	`[]` —(****), +, -, bool()
`void *`, `char *`	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 —(**)
arrays	a list or tuple of items	a <cdata>	len(), iter(), `[]` —(****), +, -

`char[]`	same as arrays, or a Python string		len(), iter(), `[]`, +, -
`wchar_t[]`	same as arrays, or a Python unicode		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

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

> *In a function declaration, as per the C standard, a `item *` argument is identical to a `item[]` argument (and `ffi.cdef()` doesn't record the difference). So when you call such a function, you can pass an argument that is accepted by either C type, like for example passing a Python string to a `char *` argument (because it works for `char[]` arguments) or a list of integers to a `int *` argument (it works for `int[]` arguments). Note that even if you want to pass a single `item`, you need to specify it in a list of length 1; for example, a `struct point_s *` argument might be passed as `[[x, y]]` or `[{'x': 5, 'y': 10}]`.*
>
> *As an optimization, the CPython version of CFFI assumes that a function with a `char *` argument to which you pass a Python string will not actually modify the array of characters passed in, and so passes directly a pointer inside the Python string object. (PyPy might in the future do the same, but it is harder because a string object can move in memory when the GC runs.)*

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

> *Note that we assume that the called functions are* not *using the Python API from Python.h. For example, we don't check afterwards if they set a Python exception. You may work around it, but mixing CFFI with `Python.h` is not recommended.*

—*(***)* `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);`.*

—*(****)* Slicing with `x[start:stop]`:

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

—*(*****)* Enums are handled like ints:

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