# Contents

1 Installation and Status .................................................. 3
   1.1 Platform-specific instructions .................................. 4

2 Examples .............................................................................. 5
   2.1 Simple example (ABI level) ......................................... 5
   2.2 Real example (API level) ............................................ 5
   2.3 Struct/Array Example ................................................ 6
   2.4 What actually happened? ............................................ 6

3 Distributing modules using CFFI .......................................... 7
   3.1 Cleaning up the __pycache__ directory ......................... 7

4 Reference ............................................................................. 9
   4.1 Declaring types and functions .................................... 9
   4.2 Loading libraries ...................................................... 9
   4.3 The verification step .................................................. 10
   4.4 Working with pointers, structures and arrays ............... 12
   4.5 Python 3 support ...................................................... 14
   4.6 An example of calling a main-like thing ..................... 14
   4.7 Function calls ......................................................... 15
   4.8 Variadic function calls .............................................. 16
   4.9 Callbacks .................................................................. 16
   4.10 Misc methods on ffi ................................................ 17
   4.11 Unimplemented features .......................................... 19
   4.12 Debugging dlopen’ed C libraries ............................... 19
   4.13 Reference: conversions ............................................ 20
   4.14 Reference: verifier ................................................... 21

5 Comments and bugs ............................................................ 23

6 Indices and tables ............................................................... 25
Foreign Function Interface for Python calling C code. The aim of this project is to provide a convenient and reliable way of calling C code from Python. The interface is based on LuaJIT’s FFI and follows a few principles:

• The goal is to call C code from Python. You should be able to do so without learning a 3rd language: every alternative requires you to learn their own language (Cython, SWIG) or API (ctypes). So we tried to assume that you know Python and C and minimize the extra bits of API that you need to learn.

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

• Work either at the level of the ABI (Application Binary Interface) or the API (Application Programming Interface). Usually, C libraries have a specified C API but often not an ABI (e.g. they may document a “struct” as having at least these fields, but maybe more). (ctypes works at the ABI level, whereas Cython and native C extensions work at the API level.)

• We 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).

• We 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.
CHAPTER 1

Installation and Status

Quick installation:

- `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 Win32. There are some Windows-specific issues left.

It supports CPython 2.6; 2.7; 3.x (tested with 3.2 and 3.3); and PyPy 2.0 beta1 or later.

Its speed is comparable to ctypes on CPython (a bit faster but a higher warm-up time). It is already faster on PyPy (1.5x-2x), but not yet much faster; stay tuned.

Requirements:

- CPython 2.6 or 2.7 or 3.x, or PyPy trunk
  - on CPython you need to build the C extension module, so you need `python-dev` and `libffi-dev` (for Windows, `libffi` is included with CFFI).
- `pycparser >= 2.06`: http://code.google.com/p/pycparser/ (Note that in old downloads of 2.08, the tarball contained an installation issue; it was fixed without changing the version number.)
- a C compiler is required to use CFFI during development, but not to run correctly-installed programs that use CFFI.
- `py.test` is needed to run the tests of CFFI.

Download and Installation:

- http://pypi.python.org/packages/source/c/cffi/cffi-0.5.tar.gz
  - Or grab the most current version by following the instructions below.
    - MD5: b163c11f68cad4371e8caeb91d81743f
    - SHA: d201e114d701eafb458ebde569acbcc5225eef
- Or get it 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 10.6)
- or you can directly import and use `cffi`, but if you don’t compile the `_cffi_backend` extension module, it will fall back to using internally `ctypes` (much slower; we recommend not to use it).
• running the tests: `py.test c/ testing/` (if you didn’t install cffi yet, you may need `python setup_base.py build` and `PYTHONPATH=build/lib.xyz.../`)

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

1.1 Platform-specific instructions

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

1.1.1 MacOS 10.6

(Thanks Juraj Sukop for this)

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

```python
include_dirs = []
```

to:

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

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

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

as described here.
2.1 Simple example (ABI level)

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

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)`.

2.2 Real example (API level)

```python
from cffi import FFI
ffi = FFI()
ffi.cdef(""
struct passwd {
   char *pw_name;
   ...;
};
struct passwd *getpwuid(int uid);
""")
C = ffi.verify(""
# include <sys/types.h>
# include <pwd.h>
""
, libraries=[]) # or a list of libraries to link with
p = C.getpwuid(0)
assert ffi.string(p.pw_name) == 'root' # on Python 3: b'root'
```

Note that the above example works independently of the exact layout of `struct passwd`. It requires a C compiler the first time you run it, unless the module is distributed and installed according to the `Distributing modules using CFFI` instructions below. See also the note about `Cleaning up the __pycache__ directory`.

You will find a number of larger examples using `verify()` in the `demo` directory.
2.3 Struct/Array Example

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

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

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

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

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

2.4 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 and global variables. 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”.

The `dlopen()` line loads libraries. C has multiple namespaces - a global one and local ones per library. In this example we load the global one (`None` as argument to `dlopen()`) which always contains the standard C library. You get as a result a `<FFILibrary>` object that has as attributes all symbols declared in the `cdef()` and coming from this library.

The `verify()` line in the second example is an alternative: instead of doing a `dlopen`, it generates and compiles a piece of C code. When using `verify()` you have the advantage that you can use “...” at various places in the `cdef()`, and the missing information will be completed with the help of the C compiler. It also does checking, to verify that your declarations are correct. If the C compiler gives warnings or errors, they are reported here.

Finally, 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 function calls should be obvious. It’s like C.
Distributing modules using CFFI

If you use CFFI and verify() in a project that you plan to distribute, other users will install it on machines that may not have a C compiler. Here is how to write a setup.py script using distutils in such a way that the extension modules are listed too. This lets normal setup.py commands compile and package the C extension modules too.

Example:

```python
from setuptools import setup
--OR--
from distutils.core import setup

# you must import at least the module(s) that define the ffi's
# that you use in your application
import yourmodule

setup(...
    zip_safe=False, # with setuptools only
    ext_modules=[yourmodule.ffi.verifier.get_extension()])
```

Warning: with setuptools, you have to say zip_safe=False, otherwise it might or might not work, depending on which verifier engine is used! (I tried to find either workarounds or proper solutions but failed so far.)

New in version 0.4: If your setup.py installs a whole package, you can put the extension in it too:

```python
setup(...
    zip_safe=False,
    ext_package='yourpackage', # but see below!
    ext_modules=[yourmodule.ffi.verifier.get_extension()])
```

However in this case you must also give the same ext_package argument to the original call to ffi.verify():

```python
ffi.verify("...", ext_package='yourpackage')
```

Usually that’s all you need, but see the Reference: verifier section for more details about the verifier object.

### 3.1 Cleaning up the __pycache__ directory

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 just completely remove the __pycache__ directory.
As a guideline: you have already seen in the above examples all the major pieces except maybe `ffi.cast()`. The rest of this documentation gives a more complete reference.

### 4.1 Declaring types and functions

`ffi.cdef(source)` parses the given C source. This should be done first. It registers all the functions, types, 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.verify()`).

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 0.4:_ `_Bool`. If not directly supported by the C compiler, this is declared with the size of `unsigned char`. Note that the effects of `<stdbool.h>` are not automatically included: you have to say `typedef _Bool bool;` in your `cdef()` if you want to use this `_Bool` with the more standard name `bool`. This is because some headers declare a different type (e.g. an enum) and also call it `bool`.
- _New in version 0.4:_ `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`.

As we will see on the verification step below, the declarations can also contain “...” at various places; these are placeholders that will be completed by a call to `verify()`.

### 4.2 Loading libraries

`ffi.dlopen(libpath, [flags])`: this function opens a shared library and returns a module-like library object. You need to use either `ffi.dlopen()` or `ffi.verify()`, documented below.
You can use the library object to call the functions previously declared by `ffi.cdef()`, 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`). Alternatively, if `libpath` is None, it returns the standard C library (which can be used to access the functions of glibc, on Linux).

This gives ABI-level access to the library: you need to have all types declared manually exactly as they were while the library was made. No checking is done. For this reason, we recommend to use `ffi.verify()` instead when possible.

Note that only functions and global variables are in library objects; 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`.

### 4.3 The verification step

`ffi.verify(source, tmpdir=.., ext_package=.., modulename=.., **kwargs)`: verifies that the current ffi signatures compile on this machine, and return a dynamic library object. The dynamic library can be used to call functions and access global variables declared by a previous `ffi.cdef()`. You don’t need to use `ffi.dlopen()` in this case.

The returned library is a custom one, compiled just-in-time by the C compiler: it gives you C-level API compatibility (including calling macros, as long as you declared them as functions in `ffi.cdef()`). This differs from `ffi.dlopen()`, which requires ABI-level compatibility and must be called several times to open several shared libraries.

On top of CPython, the new library is actually a CPython C extension module.

The arguments to `ffi.verify()` are:

- **source**: C code that is pasted verbatim in the generated code (it is *not* parsed internally). It should contain at least the necessary `#include`. It can also contain the complete implementation of some functions declared in `cdef()`: this is useful if you really need to write a piece of C code, e.g. to access some advanced macros (see the example of `getyx()` in `demo/_curses.py`).
- **sources**, `include_dirs`, `define_macros`, `undef_macros`, `libraries`, `library_dirs`, `extra_objects`, `extra_compile_args`, `extra_link_args` (keyword arguments): these are used when compiling the C code, and are passed directly to `distutils`. 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. See the `distutils` documentation for more information about the other arguments.

On the plus side, this solution gives more “C-like” flexibility:

- 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 *`.

Moreover, you can use “...” in the following places in the `cdef()` for leaving details unspecified, which are then completed by the C compiler during `verify()`:
• structure declarations: any `struct` that ends with `"...;"` 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 a `VerificationError` if it is not.

• 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 types only. In some cases you need to say that `foo_t` is not opaque, but you just don’t know any field in it; then you would use “typedef struct { ...; } foo_t;”.

• array lengths: when used as structure fields, arrays can have an unspecified length, as in “int n[];” or “int n[...];”. The length is completed by the C compiler.

• 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 macros: you can write in the `cdef` the line “#define FOO ...”, with any macro name FOO. Provided the macro is defined to be an integer value, this value will be available via an attribute of the library object returned by `verify()`. 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 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. In the case of function arguments or return type, when it is a simple integer/float type, it may be misdeclared (if you misdeclare a function `void f(long)` as `void f(int)`, it still works, but you have to call it with arguments that fit an int). But it doesn’t work any longer for more complex types (e.g. you cannot misdeclare a `int *` argument as `long *`) or in other locations (e.g. a global array `int a[5];` must not be declared `long a[5];`). CFFI considers all types listed above as primitive (so `long long a[5];` and `int64_t a[5]` are different declarations). Note the following hack to find explicitly the size of any type, in bytes:

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

Note that `verify()` is meant to call C libraries that are not using `#include <Python.h>`. The C functions are called without the GIL, and afterwards we don’t check if they set a Python exception, for example. You may work around it, but mixing CFFI with `Python.h` is not recommended.

New in version 0.4: Unions used to crash `verify()`. Fixed.

New in version 0.4: The `tmpdir` argument to `verify()` controls where the C files are created and compiled. By default it 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.)

The `ext_package` argument controls in which package the compiled extension module should be looked from. This is only useful after `distributing modules using CFFI`.

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. New in version 0.5: 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

4.3. The verification step
will be reused without further check. Be sure to have other means of clearing the tmpdir whenever you change your sources.

### 4.4 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, if supported by the backend. 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>`.

```python
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:

```python
>>> ffi.new("char *")
<cdata 'char *' owning 1 bytes>
>>> ffi.new("int *")
<cdata 'int *' owning 4 bytes>
>>> ffi.new("int[10]"
<cdata 'int[10]' owning 40 bytes>
```

Changed in version 0.2: Note that this changed from CFFI version 0.1: what used to be `ffi.new("int")` is now `ffi.new("int *").`

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

```python
global_weakkeydict = weakref.WeakKeyDictionary()

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.
```
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.

Changed in version 0.2: You will find `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)`. In version 0.1, reading a NULL pointer used to return `None`; now it returns a regular `<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()` below. 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 the above rule, dereferencing a pointer that owns a `struct` or `union` object returns a `cdata` struct or union object that “co-owns” the same memory. Thus in this case there are two objects that can keep the same memory alive. This is done for cases where you really want to have a struct object but don’t have any convenient place to keep alive the original pointer object (returned by `ffi.new()`).

Example:

```python
ffi.cdef("void somefunction(int *);")
lib = ffi.verify("#include <foo.h>"

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:

```python
>>> 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):

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

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

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

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

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

Like C, arrays of chars can also be initialized from a string, in which case a terminating null character is appended implicitly:
```python
cffi Documentation, Release 0.5

```>>> x = ffi.new("char[]", "hello")
```>>> x
<cdat\a '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 (encoding and decoding surrogates as needed 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:

```python
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):

```python
int array[1000];  # C syntax
array = ffi.new("int[1000]")  # CFFI 1st equivalent
array = ffi.new("int[1000]", 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.

### 4.5 Python 3 support

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

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

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

Imagine we have something like this:

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

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

```python
# 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:

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

Changed in version 0.3: In older versions, passing a list as the char *[] argument did not work; you needed to make an argv_keepalive and an argv in all cases.

### 4.7 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:

```python
ffi.cdef(""
      int foo(short a, int b);
""")
lib = ffi.verify("#include <foo.h>")

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

As an extension, 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!):

```python
ffi.cdef(""
      size_t strlen(const char *);
""")
C = ffi.dlopen(None)

assert C.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 int[5]. So you can pass an int * as a list of integers:

```python
ffi.cdef(""
      void do_something_with_array(int *array);
""")
lib.do_something_with_array([1, 2, 3, 4, 5])
``
CFFI supports passing and returning structs to functions and callbacks. Example (sketch):

```python
>>> ffi.cdef(""
... struct foo_s { int a, b; }
... struct foo_s function_returning_a_struct(void);
... "")
>>> lib = ffi.verify("#include <somewhere.h>"")
>>> lib.function_returning_a_struct()
<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 "...;" and completed with `verify()`; you need to declare it completely in `cdef()`.

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

### 4.8 Variadic function calls

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

```c
C.printf("hello, %d\n", 42)
```

to figure out if 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()`:

```python
C.printf("hello, %d\n", ffi.cast("int", 42))
C.printf("hello, %ld\n", ffi.cast("long", 42))
C.printf("hello, %lf\n", ffi.cast("double", 42))
C.printf("hello, %s\n", ffi.new("char[]", "world"))
```

### 4.9 Callbacks

C functions can also be viewed as cdata objects, and so can be passed as callbacks. To make new C callback objects that will invoke a Python function, you need to use:

```python
>>> def myfunc(x, y):
...     return x + y
...
>>> ffi.callback("int(int, int)", myfunc)
<cdata 'int(*)(int, int)' calling <function myfunc at 0xf757bbc4>>
```

New in version 0.4: Or equivalently as a decorator:

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

Note that you can also use a C function pointer type like "int(*)(int, int)" (as opposed to a C function type like "int(int, int)"). It is equivalent here.

Warning: like `ffi.new()`, `ffi.callback()` returns a cdata that has ownership of its C data. (In this case, the necessary C data contains the libffi data structures to do a callback.) This means that the callback can only be invoked as long as
this cdata object is alive. If you store the function pointer into C code, then make sure you also keep this object alive for as long as the callback may be invoked. (If you want the callback to remain valid forever, store the object in a fresh global variable somewhere.)

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:

```c
ffi.cdef(""
   int (*python_callback)(int how_many, int *values);
   void *const c_callback; /* pass this ptr to C routines */
"")
lib = ffi.verify(""
#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);
}
"")
lib.python_callback = python_callback
```

Windows: you can’t yet specify the calling convention of callbacks. (For regular calls, the correct calling convention should be automatically inferred by the C backend.) Use an indirection, like in the example just above.

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()`:

```python
>>> ffi.callback("int(int, int)", myfunc, error=42)
```

In all cases the exception is printed to stderr, so this should be used only as a last-resort solution.

### 4.10 Misc methods on ffi

**ffi.include(other_ffi):** includes the typedefs, structs, unions and enums defined in another FFI instance. Usage is similar to a `#include` in C, where a part of the program might include types defined in another part for its own usage. Note that the include() method has no effect on functions, constants and global variables, which must anyway be accessed directly from the `lib` object returned by the original FFI instance. *New in version 0.5.*

**ffi.errno:** the value of `errno` received from the most recent C call in this thread, and passed to the following C call, is available via reads and writes of the property `ffi.errno`. On Windows we also save and restore the `GetLastError()` value, but to access it you need to declare and call the `GetLastError()` function as usual.

**ffi.string(cdata, [maxlen]):** return a Python string (or unicode string) from the ‘cdata’. *New in version 0.3.*

- 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, or #NUMBER if the value is out of range.

`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; you can use for example `file.write()` and `file.readinto()` with such a buffer (for files opened in binary mode). (Remember that like in C, you use `array + index` to get the pointer to the index'th item of an array.)

Changed in version 0.4: 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 `buffer`):

- `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.

Changed in version 0.5: 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. Moreover buffer objects now support weakrefs to them.

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

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

New in version 0.4: `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` and `elements`.

`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 alignment of the C type. Corresponds to the `__alignof__` operator in GCC.

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

`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]"`. 

New in version 0.4:
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. New in version 0.3 (together with the fact that any cdata object can be weakly referenced).

ffi.addressof(cdata, field=None): from a cdata whose type is struct foo_s, return its “address”, as a cdata whose type is struct foo_s *. Also works on unions, but not on any other type. (It would be difficult because only structs and unions are internally stored as an indirect pointer to the data.) If field is given, returns the address of that field in the structure. The returned pointer is only valid as long as the original object is. New in version 0.4.

4.11 Unimplemented features

All of the ANSI C declarations should be supported, and some of C99. Known missing features that are GCC or MSVC extensions:

- Any __attribute__ or #pragma pack(n)
- Additional types: complex numbers, special-size floating and fixed point types, vector types, and so on. You might be able to access an array of complex numbers by declaring it as an array of struct my_complex { double real, imag; }, but in general you should declare them as struct { ...; } and cannot access them directly. This means that you cannot call any function which has an argument or return value of this type (this would need added support in libffi). You need to write wrapper functions in C, e.g. void foo_wrapper(struct my_complex c) { foo(c.real + c.imag*1j); }, and call foo_wrapper rather than foo directly.
- Thread-local variables (access them via getter/setter functions)
- Variable-length structures, i.e. whose last field is a variable-length array (work around like in C, e.g. by declaring it as an array of length 0, allocating a char[] of the correct size, and casting it to a struct pointer)
- Enum types are always int. GCC supports enums containing larger constants (unsigned int. or long long) as an extension to the C standard, but CFFI does not. Use typedef <exact type> my_enum; and then some #define foo <value>.

New in version 0.4: Now supported: the common GCC extension of anonymous nested structs/unions inside structs/unions.

4.12 Debugging dlopen’ed C libraries

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

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

export LD_TRACE_LOADED_OBJECTS=all

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

export LD VERBOSE=1
(glibc since 2.1) If set to a nonempty string, output symbol versioning information about the program if querying information about the program (i.e., either LD_TRACE_LOADED_OBJECTS has been set, or --list or --verify options have been given to the dynamic linker).

export LD_WARN=1

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

### 4.13 Reference: conversions

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

<table>
<thead>
<tr>
<th>C type</th>
<th>writing into</th>
<th>reading from</th>
<th>other operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>integers</td>
<td>an integer or anything on which int() works (but not a float!). Must be within range.</td>
<td>a Python int or long, depending on the type</td>
<td>int()</td>
</tr>
<tr>
<td>char</td>
<td>a string of length 1 or another &lt;cdata char&gt;</td>
<td>a string of length 1</td>
<td>int()</td>
</tr>
<tr>
<td>wchar_t</td>
<td>a unicode of length 1 (or maybe 2 if surrogates) or another &lt;cdata wchar_t&gt;</td>
<td>a unicode of length 1 (or maybe 2 if surrogates)</td>
<td>int()</td>
</tr>
<tr>
<td>float, double</td>
<td>a float or anything on which float() works</td>
<td>a Python float</td>
<td>float(), int()</td>
</tr>
<tr>
<td>long double</td>
<td>another &lt;cdata&gt; with a long double, or anything on which float() works</td>
<td>a &lt;cdata&gt;, to avoid loosing precision (***)</td>
<td>float(), int()</td>
</tr>
<tr>
<td>pointers</td>
<td>another &lt;cdata&gt; with a compatible type (i.e. same type or char* or void*, or as an array instead) (*)</td>
<td>a &lt;cdata&gt;</td>
<td>[ ], +, -, bool()</td>
</tr>
<tr>
<td>void *, char *</td>
<td>another &lt;cdata&gt; with any pointer or array type</td>
<td></td>
<td>[ ], +, -, bool(), and read/write struct fields</td>
</tr>
<tr>
<td>pointers to structure or union</td>
<td>same as pointers (*)</td>
<td></td>
<td>bool(), call (**)</td>
</tr>
<tr>
<td>function pointers</td>
<td>same as pointers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>arrays</td>
<td>a list or tuple of items</td>
<td>a &lt;cdata&gt;</td>
<td>len(), iter(), [ ], +, -</td>
</tr>
<tr>
<td>char[]</td>
<td>same as arrays, or a Python string</td>
<td></td>
<td>len(), iter(), [ ], +, -</td>
</tr>
<tr>
<td>wchar_t[]</td>
<td>same as arrays, or a Python unicode</td>
<td></td>
<td>len(), iter(), [ ], +, -</td>
</tr>
<tr>
<td>structure</td>
<td>a list or tuple or dict of the field values, or a same-type &lt;cdata&gt;</td>
<td>a &lt;cdata&gt;</td>
<td>read/write fields</td>
</tr>
<tr>
<td>union</td>
<td>same as struct, but with at most one field</td>
<td></td>
<td>read/write fields</td>
</tr>
<tr>
<td>enum</td>
<td>an integer, or the enum value as a string or as &quot;#NUMBER&quot;</td>
<td>the enum value as a string, or &quot;#NUMBER&quot; if out of range</td>
<td>int()</td>
</tr>
</tbody>
</table>

Changed in version 0.3: (*) Note that when calling a function, as per C, a *item* argument is identical to a *item[]* argument. So 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 foo * argument might be passed as [[field1, field2...]].

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.

Changed in version 0.3: (***) C function calls are now done with the GIL released.

New in version 0.3: (***) long double support. Such a number is passed around in a cdata object to avoid loosing precision, because a normal Python floating-point number only contains enough precision for a double. To convert it to a regular float, call float(). If you want to operate on such numbers without any precision loss, you need to define and use a family of C functions like long double add(long double a, long double b);

4.14 Reference: verifier

For advanced use cases, the Verifier class from cffi.verifier can be instantiated directly. It is normally instantiated for you by ffi.verify(), and the instance is attached as ffi.verifier.

- Verifier(ffi, preamble, tmpdir=.., ext_package='', modulename=None, tag='', **kws): instantiate the class with an FFI object and a preamble, which is C text that will be pasted into the generated C source. The value of tmpdir defaults to the directory directory_of_the_caller/__pycache__. The value of ext_package is used when looking up an already-compiled, already-installed version of the extension module. The module name is _cffi_<tag>_<hash>, unless overridden with modulename (see the warning about modulename above).

The other keyword arguments are passed directly to distutils when building the Extension object.

Verifier objects have the following public attributes and methods:

- sourcefilename: name of a C file. Defaults to tmpdir/_cffi_CRCHASH.c, with the CRCHASH part computed from the strings you passed to cdef() and verify() as well as the version numbers of Python and CFFI. Can be changed before calling write_source() if you want to write the source somewhere else.

- modulefilename: name of the .so file (or .pyd on Windows). Defaults to tmpdir/_cffi_CRCHASH.so. Can be changed before calling compile_module().

- get_module_name(): extract the module name from modulefilename.

- write_source(file=None): produces the C source of the extension module. If file is specified, write it in that file (or file-like) object rather than to sourcefilename.

- compile_module(): writes the C source code (if not done already) and compiles it. This produces a dynamic link library whose file is given by modulefilename.

- load_library(): loads the C module (if necessary, making it first; it looks for the existing module based on the checksum of the strings passed to ffi.cdef() and preamble, either in the directory tmpdir or in the directory of the package ext_package). Returns an instance of a FFILibrary class that behaves like the objects returned by ffi.dlopen(), but that delegates all operations to the C module. This is what is returned by ffi.verify().

- get_extension(): returns a distutils-compatible Extension instance.

The following are global functions in the cffi.verifier module:

- set_tmpdir(dirmame): sets the temporary directory to use instead of directory_containing_the_py_file/__pycache__. This is a global, so avoid it in production code.

- cleanup_tmpdir(tmpdir=...): cleans up the temporary directory by removing all files in it called _cffi_*.{c,so} as well as all files in the build subdirectory. By default it will clear directory_containing_the_py_file/__pycache__. This is the .py file containing the actual call to cleanup_tmpdir().
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
Indices and tables

- genindex
- search