pyBCN BARCELONA PYTHON MEETUP

cffi: C Foreign Function Interface for Python

Gerard Marull-Paretas
gerardmarull@gmail.com

23rd November 2017

Outline

Introduction

C $\leftrightarrow$ Python

`cffi`: a better approach

Real example: `ingenialink`

Conclusions

Introduction

Why bother about C?

- Still among the **top** languages in popularity as of 2017
  - #7/23 (PYPL)
  - #2/20 (TIOBE)
  - #2/10 (IEEE)

Why bother about C?

- Still among the **top** languages in popularity as of 2017
  - #7/23 (PYPL)
  - #2/20 (TIOBE)
  - #2/10 (IEEE)
- Extensively used on low level APIs (e.g. O.S. syscalls)

Why bother about C?

- Still among the **top** languages in popularity as of 2017
  - #7/23 (PYPL)
  - #2/20 (TIOBE)
  - #2/10 (IEEE)
- Extensively used on low level APIs (e.g. O.S. syscalls)
- Lots of **existing and proven** libraries

Why bother about C?

- Still among the **top** languages in popularity as of 2017
  - #7/23 (PYPL)
  - #2/20 (TIOBE)
  - #2/10 (IEEE)
- Extensively used on low level APIs (e.g. O.S. syscalls)
- Lots of **existing and proven** libraries
- Usually **good choice** for code that requires **high-performance**

Why bother about C?

- Still among the **top** languages in popularity as of 2017
  - #7/23 (PYPL)
  - #2/20 (TIOBE)
  - #2/10 (IEEE)
- Extensively used on low level APIs (e.g. O.S. syscalls)
- Lots of **existing and proven** libraries
- Usually **good choice** for code that requires **high-performance**
- Can be **wrapped to almost any language**
  - See for example libgit2 (Python, Perl, NodeJS, Go, PHP...)

Why using C libraries from Python?

- **Use existing**, proven libraries **without rewriting** them

Why using C libraries from Python?

- **Use existing**, proven libraries **without rewriting** them
- Implement **performance critical** code in C, but use it from Python

Why using C libraries from Python?

- **Use existing**, proven libraries **without rewriting** them
- Implement **performance critical** code in C, but use it from Python
- Implement a **single library in C**, then wrap it to *any* language, e.g. Python!

Why using C libraries from Python?

- **Use existing**, proven libraries **without rewriting** them
- Implement **performance critical** code in C, but use it from Python
- Implement a **single library in C**, then wrap it to *any* language, e.g. Python!
- Use **legacy libraries** with a modern language

Why using C libraries from Python?

- **Use existing**, proven libraries **without rewriting** them
- Implement **performance critical** code in C, but use it from Python
- Implement a **single library in C**, then wrap it to *any* language, e.g. Python!
- Use **legacy libraries** with a modern language
- A way to **test a C library** using Python facilities

C $\leftrightarrow$ Python

Native extensions

- The C library `libpython` allows to create **native extensions in C**

Native extensions

- The C library `libpython` allows to create **native extensions in C**
- **Any C API can be called** and thus wrapped to Python

Native extensions

- The C library `libpython` allows to create **native extensions in C**
- **Any C API can be called** and thus wrapped to Python
- The way to achieve the **best performance**

Native extensions

- The C library `libpython` allows to create **native extensions in C**
- **Any C API can be called** and thus wrapped to Python
- The way to achieve the **best performance**

but...

Native extensions

- The C library `libpython` allows to create **native extensions in C**
- **Any C API can be called** and thus wrapped to Python
- The way to achieve the **best performance**

but...

- Extensions need to be compiled

Native extensions

- The C library `libpython` allows to create **native extensions in C**
- **Any C API can be called** and thus wrapped to Python
- The way to achieve the **best performance**

but...

- Extensions need to be compiled
- Other interpreters, e.g. PyPy may not be supported

Native extensions

- The C library `libpython` allows to create **native extensions in C**
- **Any C API can be called** and thus wrapped to Python
- The way to achieve the **best performance**

but...

- Extensions need to be compiled
- Other interpreters, e.g. PyPy may not be supported
- `libpython` **API is cumbersome**, hard to learn and use

Native extensions

- The C library `libpython` allows to create **native extensions in C**
- **Any C API can be called** and thus wrapped to Python
- The way to achieve the **best performance**

but...

- Extensions need to be compiled
- Other interpreters, e.g. PyPy may not be supported
- `libpython` **API is cumbersome**, hard to learn and use
- Need to support **API differences** between Python versions

Native extensions: example

Objective: Implement a module that allows to execute system commands by wrapping the C `system` command.

```
import spam
status = spam.system("ls -l")
```

Example taken from the Python official documentation

Native extensions: example (II)

```c
#include <Python.h>

static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

Native extensions: example (III)

```c
static PyMethodDef SpamMethods[] = {
    {"system",  spam_system, METH_VARARGS,
     "Execute a shell command."},
    {NULL, NULL, 0, NULL}        /* Sentinel */
};

static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam",   /* name of module */
    NULL,     /* module documentation, may be NULL */
    -1,       /* size of per-interpreter state of the module,
                 or -1 if the module keeps state in global variables. */
    SpamMethods
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModule_Create(&spammodule);
}
```

`ctypes`

- `ctypes` allows **calling C libraries directly from Python**

`ctypes`

- `ctypes` allows **calling C libraries directly from Python**
- Based on `libffi`, used to create a *bridge* between the library and Python at run time

`ctypes`

- `ctypes` allows **calling C libraries directly from Python**
- Based on `libffi`, used to create a *bridge* between the library and Python at run time
- Included as **part of the standard library** since Python 2.5

`ctypes`

- `ctypes` allows **calling C libraries directly from Python**
- Based on `libffi`, used to create a *bridge* between the library and Python at run time
- Included as **part of the standard library** since Python 2.5
- **No need to compile**

`ctypes`

- `ctypes` allows **calling C libraries directly from Python**
- Based on `libffi`, used to create a *bridge* between the library and Python at run time
- Included as **part of the standard library** since Python 2.5
- **No need to compile**

but...

`ctypes`

- `ctypes` allows **calling C libraries directly from Python**
- Based on `libffi`, used to create a *bridge* between the library and Python at run time
- Included as **part of the standard library** since Python 2.5
- **No need to compile**

but...

- `libffi` introduces **overhead**

`ctypes`

- `ctypes` allows **calling C libraries directly from Python**
- Based on `libffi`, used to create a *bridge* between the library and Python at run time
- Included as **part of the standard library** since Python 2.5
- **No need to compile**

but...

- `libffi` introduces **overhead**
- Need to **manually declare the functions, data types**, etc.

libffi

- Compilers for high level languages generate code that follows certain conventions (e.g. calling convention)

`libffi`

- Compilers for high level languages generate code that follows certain conventions (e.g. calling convention)
- `libffi` (C) provides an interface for **calling** natively compiled functions **given information at run time instead of compile time**

`libffi`

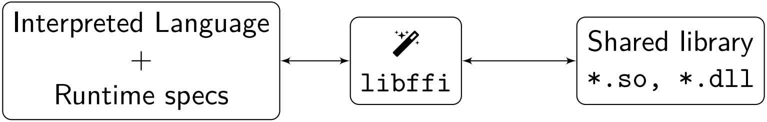
- Compilers for high level languages generate code that follows certain conventions (e.g. calling convention)
- `libffi` (C) provides an interface for **calling** natively compiled functions **given information at run time instead of compile time**
- Can produce a function that can accept and decode any combination of arguments defined at runtime

`libffi`

- Compilers for high level languages generate code that follows certain conventions (e.g. calling convention)
- `libffi` (C) provides an interface for **calling** natively compiled functions **given information at run time instead of compile time**
- Can produce a function that can accept and decode any combination of arguments defined at runtime
- Often used a a **bridge between compiled and interpreted languages**

libffi

- Compilers for high level languages generate code that follows certain conventions (e.g. calling convention)
- `libffi` (C) provides an interface for **calling** natively compiled functions **given information at run time instead of compile time**
- Can produce a function that can accept and decode any combination of arguments defined at runtime
- Often used a a **bridge between compiled and interpreted languages**



ctypes: example

Objective: Traverse a directory content using `readdir`, found in glibc.

```
struct dirent *readdir(DIR *dirp);
```

The `readdir()` function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp. It returns NULL on reaching the end of the directory stream or if an error occurred. On Linux, the dirent structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all file system types */
    char           d_name[256]; /* filename */
};
```

Example taken from the Eli Bendersky's website

ctypes: example (II)

```python
import ctypes as ct

# load library
# None as libc is already loaded, could be explicitely loaded
lib = ct.CDLL(None)

# declare the types needed for readdir.
class DIRENT(ct.Structure):
    _fields_ = [('d_ino', ct.c_long),
                ('d_off', ct.c_long),
                ('d_reclen', ct.c_ushort),
                ('d_type', ct.c_ubyte),
                ('d_name', ct.c_char * 256)]

DIR_p = ct.c_void_p
DIRENT_p = ct.POINTER(DIRENT)
```

ctypes: example (III)

```
# declare needed functions
readdir = lib.readdir
readdir.argtypes = [DIR_p]
readdir.restype = DIRENT_p

opendir = lib.opendir
opendir.argtypes = [ct.c_char_p]
opendir.restype = DIR_p

closedir = lib.closedir
closedir.argtypes = [DIR_p]
closedir.restype = ct.c_int
```

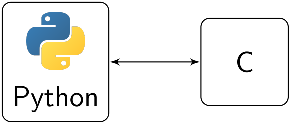
ctypes: example (IV)

```
# open directory
dir = opendir(b'/tmp')
if not dir:
    raise RuntimeError('opendir failed')

# traverse directory
dirent = readdir(dir)
while dirent:
    print(dirent.contents.d_name)
    dirent = readdir(dir)

# close directory
closedir(dir)
```

`cffi`: a better approach

What is `cffi`?

`cffi` is a C Foreign Function Interface for Python. It allows to interact with **almost any C code from Python**, based on C-like **declarations** that you can often **copy-paste from header files** or documentation.



Better with an example

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

ABI vs. API levels

What really makes `cffi` different from `ctypes`, apart from easier declarations, is that it **offers two access levels**: **ABI** and **API**.

But what does it mean?

ABI level

- Calls to your C library go through `libffi`

ABI level

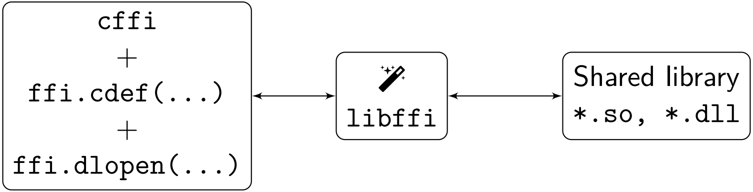
- Calls to your C library go through `libffi`
- It can be seen as *another* `ctypes`

ABI level

- Calls to your C library go through `libffi`
- It can be seen as *another* `ctypes`
- It does **not require any pre-compilation** step

ABI level

- Calls to your C library go through `libffi`
- It can be seen as *another* `ctypes`
- It does **not require any pre-compilation** step



cffi ABI, previous example

```python
from cffi import FFI

ffi = FFI()
ffi.cdef("""
    typedef void DIR;
    typedef long ino_t;
    typedef long off_t;

    struct dirent {
        ino_t          d_ino;       /* inode number */
        off_t          d_off;       /* offset to the next dirent */
        unsigned short d_reclen;    /* length of this record */
        unsigned char  d_type;      /* type of file; not supported
                                       by all file system types */
        char           d_name[256]; /* filename */
    };

    DIR *opendir(const char *name);
    struct dirent *readdir(DIR *dirp);
    int closedir(DIR *dirp);
""")
```

`cffi` ABI, previous example (II)

```
# load library
# None as libc is already loaded, could be explicitely loaded
lib = ffi.dlopen(None)

# open directory
dir = lib.opendir(b'/tmp')
if not dir:
    raise RuntimeError('opendir failed')

# traverse directory
dirent = lib.readdir(dir)
while dirent:
    print(ffi.string(dirent.d_name))
    dirent = lib.readdir(dir)

# close directory
lib.closedir(dir)
```

API level

- `cffi` **automatically creates a native C extension** based on your declarations

API level

- `cffi` **automatically creates a native C extension** based on your declarations
- The **native extension is linked with your library**

API level

- `cffi` **automatically creates a native C extension** based on your declarations
- The **native extension is linked with your library**
- **Code similar to ABI** mode (just some extras)

API level

- `cffi` **automatically creates a native C extension** based on your declarations
- The **native extension is linked with your library**
- **Code similar to ABI** mode (just some extras)
- No `libffi` magic required!

API level

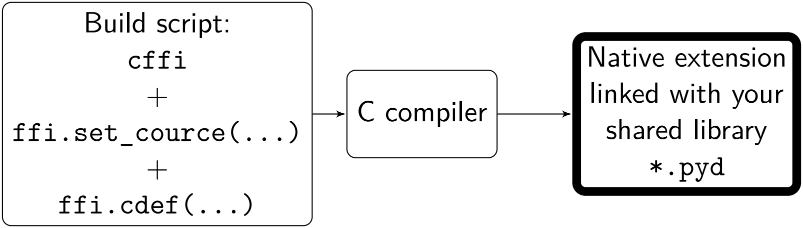
- `cffi` **automatically creates a native C extension** based on your declarations
- The **native extension is linked with your library**
- **Code similar to ABI** mode (just some extras)
- No `libffi` magic required!
- It **requires a pre-compilation** step before it can be used

API level

- `cffi` **automatically creates a native C extension** based on your declarations
- The **native extension is linked with your library**
- **Code similar to ABI** mode (just some extras)
- No `libffi` magic required!
- It **requires a pre-compilation** step before it can be used
- `cffi` integrates with `setuptools` to ease build process

API level

- `cffi` **automatically creates a native C extension** based on your declarations
- The **native extension is linked with your library**
- **Code similar to ABI** mode (just some extras)
- No `libffi` magic required!
- It **requires a pre-compilation** step before it can be used
- `cffi` integrates with `setuptools` to ease build process



cffi API, previous example

We first create the extension build script:

```
from cffi import FFI

ffibuilder = FFI()
ffibuilder.set_source("_example",
        """ /* passed to the compiler */
            #include <dirent.h>
        """,
        libraries=[] # can link to any library)

ffibuilder.cdef("""
    typedef void DIR;
    typedef long ino_t;
    typedef long off_t;

    struct dirent {
        ino_t          d_ino;       /* inode number */
    ... // truncated
""")

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

cffi API, previous example (II)

Which, when built, can then be used like this:

```
from _example import lib, ffi

# open directory
dir = lib.opendir(b'/tmp')
if not dir:
    raise RuntimeError('opendir failed')

# traverse directory
dirent = lib.readdir(dir)
while dirent:
    print(ffi.string(dirent.d_name))
    dirent = lib.readdir(dir)

# close directory
lib.closedir(dir)
```

But not only that...

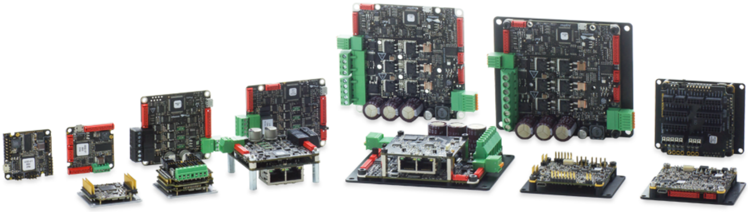
You can even **create extensions where no existing library is called**, e.g. to **implement** some algorithms **directly in C**. Or even **a mix**!

```python
# file "example_build.py"

from cffi import FFI
ffibuilder = FFI()

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

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

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

Real example: `ingenialink`

Background

Ingenia: producer of advanced **motion controllers**, with customers in the fields of **automation**, **robotics**, etc.



What we wanted/needed

- A **multiplatform library to communicate** with Ingenia's servo drives and perform some **motion control** tasks

What we wanted/needed

- A **multiplatform library to communicate** with Ingenia's servo drives and perform some **motion control** tasks
- A library for our **configuration software**, but at the same time for **testing** or **automating** tasks

What we wanted/needed

- A **multiplatform library to communicate** with Ingenia's servo drives and perform some **motion control** tasks
- A library for our **configuration software**, but at the same time for **testing** or **automating** tasks
- A library that our **customers** can use for their typical applications

What we wanted/needed

- A **multiplatform library to communicate** with Ingenia's servo drives and perform some **motion control** tasks
- A library for our **configuration software**, but at the same time for **testing** or **automating** tasks
- A library that our **customers** can use for their typical applications
- A **single code base**, cannot maintain too many libraries

What we wanted/needed

- A **multiplatform library to communicate** with Ingenia's servo drives and perform some **motion control** tasks
- A library for our **configuration software**, but at the same time for **testing** or **automating** tasks
- A library that our **customers** can use for their typical applications
- A **single code base**, cannot maintain too many libraries
- An **easy to learn language**, with a **strong community** and lots of resources

The reality

- We love Python, but **not everybody in the world does**

The reality

- We love Python, but **not everybody in the world does**
- **Automation industry** is really **conservative**, non-friendly to changes

The reality

- We love Python, but **not everybody in the world does**
- **Automation industry** is really **conservative**, non-friendly to changes
- Most of our **customers are not up to date** with today's development practices

The reality

- We love Python, but **not everybody in the world does**
- **Automation industry** is really **conservative**, non-friendly to changes
- Most of our **customers are not up to date** with today's development practices
- Customers **still with PLC (sigh), VB, VB.NET, Win32-like APIs** and similar crap

Our approach

- Create a **library in C**, **object-oriented**, targetting **Windows, Linux and macOS**

Our approach

- Create a **library in C**, **object-oriented**, targetting **Windows, Linux and macOS**
  - Allows to **speed-up and fine-tune critical areas**

Our approach

- Create a **library in C**, **object-oriented**, targetting **Windows, Linux and macOS**
  - Allows to **speed-up and fine-tune critical areas**
  - Can create **wrappers to virtually any language** if needed, not tied to Python

Our approach

- Create a **library in C**, **object-oriented**, targetting **Windows, Linux and macOS**
  - Allows to **speed-up and fine-tune critical areas**
  - Can create **wrappers to virtually any language** if needed, not tied to Python
  - **Core code** (e.g. protocol, motion) in a **single code base**

Our approach

- Create a **library in C**, **object-oriented**, targetting **Windows, Linux and macOS**
  - Allows to **speed-up and fine-tune critical areas**
  - Can create **wrappers to virtually any language** if needed, not tied to Python
  - **Core code** (e.g. protocol, motion) in a **single code base**
- Use `cffi` to create a *native* Python extension

Our approach

- Create a **library in C**, **object-oriented**, targetting **Windows, Linux and macOS**
  - Allows to **speed-up and fine-tune critical areas**
  - Can create **wrappers to virtually any language** if needed, not tied to Python
  - **Core code** (e.g. protocol, motion) in a **single code base**
- Use `cffi` to create a *native* Python extension
- Create a **class-based API in Python** on top of the native extension

Our approach

- Create a **library in C**, **object-oriented**, targetting **Windows, Linux and macOS**
  - Allows to **speed-up and fine-tune critical areas**
  - Can create **wrappers to virtually any language** if needed, not tied to Python
  - **Core code** (e.g. protocol, motion) in a **single code base**
- Use `cffi` to create a *native* Python extension
- Create a **class-based API in Python** on top of the native extension

https://github.com/ingenialink
https://github.com/ingenialink-python

How the extension is built

`setup.py` → `cffi_modules`

Build script: `ingenialink_build.py`

`git clone` C library (and dependencies)
Build libraries as **static, but PIC**
Extract **declarations automatically from headers**
Invoke `cffi`, linking with the built libraries

Native extension with `ingenialink` **embedded**

Problems we have faced

- Dealing with **multiplatform libraries is not that nice in C**

Problems we have faced

- Dealing with **multiplatform libraries is not that nice in C**
- Python GC (*Garbage Collector*) may **not necessarily destroy objects in order when exiting** and we have object dependencies (e.g. `Servo` depends on `Network` resources)

Problems we have faced

- Dealing with **multiplatform libraries is not that nice in C**
- Python GC (*Garbage Collector*) may **not necessarily destroy objects in order when exiting** and we have object dependencies (e.g. `Servo` depends on `Network` resources)
- Linux binary wheels **need to be built using an ancient CentOS image**, which does not come with one of our dependencies (part of `systemd`)

Example

```c
#include <ingenalink/ingenialink.h>

double position;

il_net_t *net = il_net_create("/dev/ttyACM0");
il_servo_t *servo = il_servo_create(net, ID, TIMEOUT);

il_servo_read(servo, &IL_REG_POS_ACT, &position);
printf("Position: %.2f\n", position);

il_servo_destroy(servo);
il_net_destroy(net);
```

becomes...

```python
import ingenalink as il
from ingenalink import regs

net = il.Network('/dev/ttyACM0')
servo = il.Servo(net, ID)

position = servo.read(regs.POS_ACT)
print('Position: {.2f}'.format(position))
```

Application Example

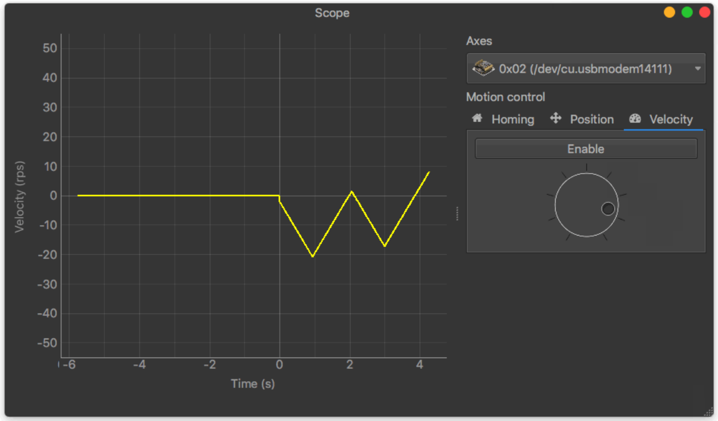


Figure: Application example (uses PyQt/PySide and pyqtgraph)

Conclusions

Conclusions

- `cffi` allows to create ***native-like* extensions with no effort**

Conclusions

- `cffi` allows to create ***native-like*** **extensions with no effort**
- **Data types** and **function declarations** can be ***copy-pasted*** **from headers** or even directly extracted

Conclusions

- `cffi` allows to create ***native-like* extensions with no effort**
- **Data types** and **function declarations** can be ***copy-pasted* from headers** or even directly extracted
- `cffi` API level extensions require compilation, although binary **wheels** remove the requirement to the end-user

Conclusions

- `cffi` allows to create ***native-like* extensions with no effort**
- **Data types** and **function declarations** can be ***copy-pasted* from headers** or even directly extracted
- `cffi` API level extensions require compilation, although binary **wheels** remove the requirement to the end-user
- `cffi` can be used in **ABI** mode (like `ctypes`) or **API** mode (like native extensions) seamlessly

Conclusions

- `cffi` allows to create ***native-like*** **extensions with no effort**
- **Data types** and **function declarations** can be ***copy-pasted*** **from headers** or even directly extracted
- `cffi` API level extensions require compilation, although binary **wheels** remove the requirement to the end-user
- `cffi` can be used in **ABI** mode (like `ctypes`) or **API** mode (like native extensions) seamlessly
- With `cffi` we can **directly define C functions** for critical parts with no need to create a shared library

Conclusions

- `cffi` allows to create ***native-like* extensions with no effort**
- **Data types** and **function declarations** can be ***copy-pasted* from headers** or even directly extracted
- `cffi` API level extensions require compilation, although binary **wheels** remove the requirement to the end-user
- `cffi` can be used in **ABI** mode (like `ctypes`) or **API** mode (like native extensions) seamlessly
- With `cffi` we can **directly define C functions** for critical parts with no need to create a shared library
- We can create a **single code base in C**, then wrap to *any language*

THANK YOU!

Questions?

https://github.com/gmarull/pybcn-cffi