CFFI for Python

Calling C functions without hassle

Henning Pridöhl
<8pridoeh@inf>

7. November 2013

Outline

Outline

What is CFFI

CFFI is a Foreign Function Interface for calling C code.

Advantages:

- Not another language to learn. If you know C and Python, you just need a very simple API.
- Supports CPython and *PyPy*.
- All Python-related logic is in Python.
- Can work at the ABI and the *API* level.
- Written by people[1] of the PyPy core team :-)

Installation:

```
pip install cffi
```

[1] Armin Rigo and Maciej Fijalkowski

A side note to “ABI vs. API”

Application Binary Interface: Describes how to data has to be put into memory/registers for the machine to execute the function.

Application Programming Interface: Describes what the programmer has to do to call the function.

A side note to "ABI vs. API": An example

```
#define LINEMODE_SOLID 1
#define LINEMODE_DOTTED 2

struct image_t {
    int width;
    int height;
    uint8_t *data;
    int _internal_stuff;
    double _more_internal_stuff;
};

void drawLine(struct image_t, int x0, int x1,
              int y0, int y1, int mode);
```

Most used functions of the API

`FFI.cdef(code)` – Just paste your C header snippets here.

`FFI.verify(code)` – Real C code to check/complete definitions.

`FFI.dlopen(library)` – Open some library for dynamic linking.

`FFI.cast(type, value)` – Like a C-cast

`FFI.new(type, value)` – Allocate some structures.

Okay, let's see some code!

Outline

Simple example

```
from cffi import FFI

ffi = FFI()
ffi.cdef("""
int add(int, int);
""")
C = ffi.verify("""
int add(int a, int b) {
    return a + b;
}
""")
print C.add(21, 21)
```

A simple image library

```
#define LINEMODE_SOLID 1
#define LINEMODE_DOTTED 2

struct image_t {
    int width;
    int height;
    uint8_t *data;
    int _internal_stuff;
    double _more_internal_stuff;
};

void drawLine(struct image_t, int x0, int x1,
              int y0, int y1, int mode);
```

More complex example

```
ffi = FFI()
ffi.cdef("""
#define LINEMODE_SOLID ...;
struct image_t {
    int width;
    int height;
    uint8_t *data;
    ...
};
void drawLine(struct image_t, int x0, int x1,
              int y0, int y1, int mode);
""")
C = ffi.verify("#include <image/image.h>",
               libraries=['image']);
img = ffi.new("image_t")
img.width, img.height = 100, 100
img.data = ffi.new("uint8_t[]", img.width * img.height)
C.drawLine(img, 0, 0, 100, 100, C.LINEMODE_SOLID);
```

More complex example in ctypes

```
from ctypes import *

libimg = CDDL("libimage.so")
LINEMODE_SOLID = 1

class(Image_t):
  _fields_ = [("width", c_int),
              ("height", c_int),
              ("data", POINTER(c_uint8)),
              ("_internal_stuff", c_int),
              ("_more_internal_stuff", c_double)]
dataType = c_uint8 * w * h
img = Image_t(100, 100, dataType(), 0, 0.0)
drawLine = libimg.drawLine
drawLine.restype = None
drawLine.argtypes = [Image_t, c_int, c_int, c_int, c_int, c_int]
drawLine(img, 0, 0, 100, 100, LINEMODE_SOLID)
```

Callbacks

```
ffi = FFI()

@ffi.callback("int(int, int)")
def add(x, y):
    return x + y

ffi.cdef("""
    int reduce(int(*)(int, int) fn, int* lst, int len);
""")
C = ffi.dlopen("./reduce.so")

lst = [1, 2, 3]
print C.reduce(add, lst, len(lst))
```

Create BTRF Snapshot

```
ffi = cffi.FFI()
ffi.cdef("""
    #define BTRFS_IOC_SNAP_CREATE_V2 ...
    struct btrfs_ioctl_vol_args_v2 {
        int64_t fd;
        char name[];
        ...;
    };
""")
v = ffi.verify("#include <btrfs/ioctl.h>")
target = os.open(target_directory, os.O_DIRECTORY)
source = os.open(source_directory, os.O_DIRECTORY)
args = ffi.new('struct btrfs_ioctl_vol_args_v2 *')
args.name = 'Mein Snapshot'
args.fd = source
args_buffer = ffi.buffer(args)
fcntl.ioctl(target, v.BTRFS_IOC_SNAP_CREATE_V2, args_buffer)
```

Complete define/struct

```
#define BTRFS_IOC_SNAP_CREATE_V2 _IOW(BTRFS_IOCTL_MAGIC, \
        23, struct btrfs_ioctl_vol_args_v2)
struct btrfs_ioctl_vol_args_v2 {
    __s64 fd;
    __u64 transid;
    __u64 flags;
    union {
        struct {
            __u64 size;
            struct btrfs_qgroup_inherit __user
                                *qgroup_inherit;
        };
        __u64 unused[4];
    };
    char name[BTRFS_SUBVOL_NAME_MAX + 1];
}
```

Outline

Numpy memory layout

A Numpy array is basically a struct with the following information:

- Type (e. g. uint8, float, double ...)
- Shape (e. g 2x3)
- Flat array of values

The method `reshape` just changes the *view* of the data.

Example:

```
a = numpy.arange(6, dtype=numpy.uint8).reshape(2, 3)
```

Internal data representation:

```
uint8_t data[6] = {0, 1, 2, 3, 4, 5}
```

View on the data:

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$$

Using CFFI with NumPy – C code

```
from cffi import FFI

ffi = FFI()
ffi.cdef("""
    void add1(float *in, float *out, int width, int height);
""")
C = ffi.verify("""
void add1(float *in, float *out, int width, int height) {
    int i = 0;
    for(int y = 0; y < height; ++y) {
        for(int x = 0; x < width; ++x, ++i) {
            out[i] = in[i] + 1;
        }
    }
}
""")
```

Using CFFI with NumPy – Python part

```
w, h = 640, 480
img_src = np.ones(w*h, dtype=np.float32).reshape(h, w)
img_dest = np.zeros_like(a)
p_img_src = ffi.cast("float *", img_src.ctypes.data)
p_img_dest = ffi.cast("float *", img_dest.ctypes.data)
C.add1(p_img_src, p_img_dest,
       img_src.shape[1], img_src.shape[0])
```

Outline

Summary

- CFFI allows to call C functions without hassle.
- API is easy to learn, if you know C and Python.
- Supports *API access* of C functions, for much more safety compared to ABI access.
- Integration with NumPy is easy
- Just stop using `ctypes` and enjoy your life :-)