CFFI and PyPy

Armin Rigo

Swiss Python Summit 2016

Feb 2016

CFFI

- successful project according to PyPI
- 3.4 million downloads last month
- total 19.2 millions, 27th place on *pypi-ranking.info*
  - Django is 28th
- some high-visibility projects have switched to it (Cryptography)

PyPy

- harder to say, but probably not so successful
- more later

CFFI

CFFI

- call C from Python
- CFFI = C Foreign Function Interface
- shares ideas from Cython, ctypes, and LuaJIT's FFI

CFFI demo

```
$ man getpwuid

SYNOPSIS
     #include <sys/types.h>
     #include <pwd.h>

     struct passwd *getpwnam(const char *);
```

CFFI demo

```
.
.
.
The passwd structure is defined in <pwd.h>
as follows:

struct passwd {
    char *pw_name; /* username */
    char *pw_passwd; /* user password */
    uid_t pw_uid; /* user ID */
.
.
.
```

CFFI demo

```
from cffi import FFI
ffi = cffi.FFI()

ffi.cdef("""
    typedef int...  uid_t;
    struct passwd {
        uid_t pw_uid;
        ...;
    };
    struct passwd *getpwnam(const char *);
""")
```

CFFI demo

```
ffi.set_source("_pwuid_cffi", """
     #include <sys/types.h>
     #include <pwd.h>
""")

ffi.compile()
```

... and put that in pwuid_build.py

CFFI demo

```
python pwuid_build.py
```

creates _pwuid_cffi.so

CFFI demo

```
from _pwuid_cffi import lib
print lib.getpwnam("username").pw_uid
```

CFFI demo

```
from _pwuid_cffi import ffi, lib
```

- `lib` gives access to all functions from the cdef
- `ffi` gives access to a few general helpers

ffi.cdef()

```
ffi.cdef("""
     int foo1(int a, int b);

     typedef ...  Window;
     Window *make_window(int w, int h);
     void hide_window(Window *);
""")
```

ffi.new()

```
>>> p = ffi.new("char[]", "Some string")
>>> p
<cdata 'char[]' owning 12 bytes>

>>> p[1]
'o'

>>> q = lib.getpwnam(p)
>>> q
<cdata 'struct passwd *' 0x12345678>

>>> q.pw_uid
500
```

ffi.cast()

```
>>> p = lib.getpwnam("root")
>>> p
<cdata 'struct passwd *' 0x12345678>

>>> ffi.cast("void *", p)
<cdata 'void *' 0x12345678>

>>> ffi.cast("long", p)
305419896
>>> hex(_)
0x12345678
```

ffi.new_handle()

```
>>> h1 = ffi.new_handle(some_object)
>>> h1
<cdata 'void *' handle to
                    <X object at 0x123456>>
>>> lib.save_away(h1)

>>> h2 = lib.fish_again()
>>> h2
<cdata 'void *' 0x87654321>

>>> ffi.from_handle(h2)
<X object at 0x123456>
```

ffi.string()

```
>>> p
<cdata 'struct passwd *' 0x12345678>

>>> p.pw_uid
500

>>> p.pw_name
<cdata 'char *' 0x5234abcd>

>>> ffi.string(p.pw_name)
"username"
```

CFFI

- supports more or less the whole C
- there is more than my short explanations suggests
- read the docs: `http://cffi.readthedocs.org/`

PyPy

PyPy

- a Python interpreter
- different from the standard, which is CPython
- main goal of PyPy: speed

PyPy

```
$ pypy
Python 2.7.10 (5f8302b8bf9f, Nov 18 2015,
[PyPy 4.0.1 with GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or
>>>> 2+3
5
>>>>
```

PyPy

- run `pypy my_program.py`
- starts working like an interpreter
- then a Just-in-Time Compiler kicks in
- generate and execute machine code from the Python program
- good or great speed-ups for the majority of long-running code

PyPy

- different techniques than CPython also for "garbage collection"
- works very well (arguably better than CPython's reference counting)

PyPy: Garbage Collection

- "**moving,** generational, incremental GC"
- objects don't have reference counters
- allocated in a "nursery"
- when nursery full, find surviving nursery objects and move them out
- usually work on nursery objects only (fast), but rarely also perform a full GC

PyPy: C extensions

- PyPy works great for running Python
- less great when there are CPython C extension modules involved

PyPy: C extensions

- not directly possible: we have moving, non-reference-counted objects, and the C code expects non-moving, reference-counted objects

PyPy: C extensions

- PyPy has still some support for them, called its `cpyext` module
- similar to IronPython's Ironclad
- emulate all objects for C extensions with a shadow, non-movable, reference-counted object

PyPy: C extensions

- `cpyext` is slow
- `cpyext` is actually *really, really* slow
  - but we're working on making it *only* slow

PyPy: C extensions

- `cpyext` will "often" work, but there are a some high-profile C extension modules that are not supported so far
- notably, `numpy`
- (it is future work)

PyPy: ad

- but, hey, if you need performance out of Python and don't rely critically on C extension modules, then give PyPy a try
  - typical area where it works well: web services

CPython C API: the problem

- CPython comes with a C API
- very large number of functions
- assumes objects don’t move
- assumes a "reference counting" model

CPython C API

- actually, the API is some large subset of the functions inside CPython itself

CPython C API

- easy to use from C
- historically, part of the success of Python

CPython C API

- further successful tools build on top of that API:
  - SWIG
  - Cython
  - and other binding generators
  - now CFFI

CFFI

- but CFFI is a bit different
  - it does not expose any part of the CPython C API
  - everything is done with a minimal API on the `ffi` object which is closer to C
    - `ffi.cast()`, `ffi.new()`, etc.
  - that means it can be directly ported

CFFI and PyPy

- we have a PyPy version of CFFI
- the demos I have given above work equally well on CPython or on PyPy
- (supporting PyPy was part of the core motivation behind CFFI)

CFFI: performance

- in PyPy, JIT compiler speeds up calls, so it's very fast
- in CPython, it doesn't occur, but it is still reasonable when compared with alternatives
- main issue is that we write more code in Python with CFFI, which makes it slower on CPython---but not really on PyPy

CFFI: summary

- call C from Python
- works natively on CPython and on PyPy
  - and easy to port to other Python implementations
- supports CPython 2.6, 2.7, 3.2 to 3.5, and is integrated with PyPy

CFFI

- independent on the particular details of the Python implementation
  - using CFFI, you call C functions and manipulate C-pointer-like objects directly from Python
  - you do in Python all logic involving Python objects
  - there are no (official) ways around this API to call the CPython C API, and none are needed

CFFI

- two reasons to switch to it :-)
  - easy and cool
  - better supported on non-CPython implementations

CFFI: latest news

- support for "embedding" Python inside some other non-Python program
  - now you really never need the CPython C API any more

CFFI

http://cffi.readthedocs.org/