# C/C++ for Python's Use: You Wrap It

Yung-Yu Chen

yyc@solvcon.net

The Happy Folk

December 26, 2013

# Python is Slow, Way Too Slow

- The rule of thumb: Python is two orders of magnitude slower than C.
  - 10 milliseconds in C: 1 second in Python.
  - 8 hours in C: a month in Python.
- It's not OK to spend a month on an over-night job.
  - But we just can't lose the productivity by Python.
  - So we want to connect Python with C (and/or C++).
  - Another case: Python is the thin wrapper of the underlying system.

# Ways to Wrapper

- Cython for C and Boost.Python for C++.
- There are other options:
  - ctypes and/or cffi.
  - SWIG.
  - http://www.ohloh.net/p/pygccxml.
  - ... and many that we don't talk about.

# In Conclusion ...

- If you don't need to deal with C++. Congratulations. You can just use Cython without worrying anything.
- If unfortunately you are working with C++. Go Boost.Python.
- `http://docs.python.org/2/c-api/index.html`
    - That is, use Python C API to make the low-level code available/callable in the Python interpreter.
    - Nothing stops you from wrapping by your bare hands with the C API.
    - But it's just an unpleasant approach (and error-prone).

# Let's Start with the Bad Boy

- C++ and Python both rely on their own OO systems.
    - Both systems are powerful but substantially different.
- Boost.Python maps everything between C++ and Python.
    - Functions, classes, exceptions, containers, iterators, etc.
    - It's *like* writing Python extension with C++.

# Official Boost.Python Hello World

```cpp
#include <boost/python.hpp>
char const* greet() {
    return "hello, world";
}
BOOST_PYTHON_MODULE(hello_ext) {
    using namespace boost::python;
    def("greet", greet);
}
```

- `#include <boost/python.hpp>`: turn on Boost.Python.
- `BOOST_PYTHON_MODULE`: make a Python module.
- `def("greet", greet)`: wrap a function.

Done.

# Boost.Python Thinks from Bottom up

- It is a C++ library (part of boost). It operates around a fundamental C++ code base.
- To use the comprehensive wrapping capabilities, one has to write the code himself.
  - No code generation other than C++ template.
  - C++ only and only C++.

# Exceptions

```cpp
#include <boost/python.hpp>
#include <boost/python/errors.hpp>
void translate(const SomeException &err) {
    PyErr(SetString(PyExc_MemoryError , err.what());
}
BOOST_PYTHON_MODULE(error_ext) {
    using namespace boost::python;
    register_exception_translator<SomeException>(translate);
}
```

- #include <boost/python/errors.hpp>
- translate()
- register_exception_translator

# Iterators

```
#include <boost/python.hpp>
#include <list>
struct Iterable {
  typedef std::list<int>::iterator iterator;
  iterator begin() { m_data.begin(); }
  iterator end() { m_data.end(); }
  std::list<int> m_data;
}
BOOST_PYTHON_MODULE(iter_ext) {
  using namespace boost::python;
  class_<Iterable>("Iterable", init<>())
  .add_property("iter", range(Iterable::begin, Iterable::end))
  .def("__iter__", range(Iterable::begin, Iterable::end))
  ;
}
```

- `boost::python::range`: iterate from begin to end.
- Alternately, `__getitem__` and `__len__` can help.

# Bi-Directional Conversions

- Python has GC while C++ doesn't: conversions of objects usually need to construct intermediate objects.
    - It's more complex when a container is involved.
- What to look for:
    - `to_python_converter`: take C++ object and expose it to Python.
    - `boost::python::converter::registry`: take Python object and give it to C++.
- When exchanging containers or large objects, using `boost::shared_ptr` as a holding type helps.

# Obvious Caveats

- We need basic understanding about the CPython memory management.
  - Ownership of references for reference counting:
    `http://docs.python.org/2/c-api/intro.html#reference-count-details`
- Be very careful about passing objects across the interpreter barrier.
  - Do not return intermediates, although Boost.Python checks for invalid returns.
  - Some return policy copies objects. It can hit the performance for large objects, or give wrong results for containers.

# What Is Cython?

- Cython generates C code from Python.
    - We need to use its command-line (`cython`) like a compiler.
- The generated code contains Python C API calls.
    - Can interface to both the existing C and C++ code.
- Some constructs can be translated into native C, and get utmost speed-up.
- `http://www.cython.org/`.

# Cython is a Language

- Cython extends Python. It is a superset of Python.
  - When compiling `.py` files, of course only Python syntax can be used.
- If Cython-specific syntax is used, the source code can no longer be run by a vanilla Python VM.
  - It should be saved as a `.pyx` file and compiled by Cython compiler to binary.
- The Cython add-on helps to generate faster code and adapt to various libraries.
- It's a top-down tool, in contrast to Boost.Python.

# Call C from Cython/Python

Let's say we have a C function `caction()`:

```c
void caction(double arr0[][1000], double arr1[][1000]) {
    for (int it=1; it<999; it++) {
        for (int jt=1; jt<999; jt++) {
            arr0[it][jt] += arr1[it-1][jt-1];
            arr0[it][jt] += arr1[it-1][jt  ];
            arr0[it][jt] += arr1[it-1][jt+1];
            arr0[it][jt] += arr1[it  ][jt+1];
            arr0[it][jt] += arr1[it+1][jt+1];
            arr0[it][jt] += arr1[it+1][jt  ];
            arr0[it][jt] += arr1[it+1][jt-1];
            arr0[it][jt] += arr1[it  ][jt-1];
        };
    };
};
```

# Cython Let You Call C

```
import numpy as np
cimport numpy as cnp
cdef extern:
    void caction(double* arr0, double* arr1)
def action():
    cdef cnp.ndarray[double, ndim=2] arr0 = np.empty(
        [1000,1000], dtype='float64')
    arr0.fill(0)
    cdef cnp.ndarray[double, ndim=2] arr1 = np.empty(
        [1000,1000], dtype='float64')
    arr1.fill(1)
    caction(&arr0[0,0], &arr1[0,0])
    assert 7968032 == arr0.sum()
```

- `cdef extern`: tells Cython that we have a C function to be used.
- Then we just use it. Cython takes care of the Python C API boilerplates.

# Useful Syntax for Wrapping

- `cdef extern from "header.h":`
  - Allows Cython to check the declarations from outside headers.
- `cdef public:`
  - Allows Cython to generates C header files for inclusion in other C code.

# Cython Is for Number-Crunching

Consider the Python version "action":

```
import numpy as np
def action():
    arr0 = np.empty([1000,1000], dtype='float64')
    arr0.fill(0)
    arr1 = np.empty([1000,1000], dtype='float64')
    arr1.fill(1)
    cdef int it = 1
    cdef int jt
    while it < 999:
        jt = 1
        while jt < 999:
            arr0[it, jt] += arr1[it-1, jt-1]
            arr0[it, jt] += arr1[it-1, jt  ]
            arr0[it, jt] += arr1[it-1, jt+1]
            arr0[it, jt] += arr1[it  , jt+1]
            arr0[it, jt] += arr1[it+1, jt+1]
            arr0[it, jt] += arr1[it+1, jt  ]
            arr0[it, jt] += arr1[it+1, jt-1]
            arr0[it, jt] += arr1[it  , jt-1]
            jt += 1
        it += 1
    assert 7968032 == arr0.sum()
```

# Let's See the Speed-Up

- C version runs 51.4 milliseconds; Python version runs 5.74 seconds.
- 5.74 sec / 51.4 msec = 106 times faster.
- If our code is for number-crunching (calculation takes the majority of time), Python + Cython will deliver C-like performance.

# Technique #1: Static Typing

- Just prefix "cdef int" to our counter:

```
def action():
    cdef int it = 0
    while it < 1000000:
        it += 1
    print it
```

- Then 16 times faster. (3.01 msec vs 49 msec)

```
> python -m timeit -s 'from example1d import
   action' 'action()'
100 loops, best of 3: 3.01 msec per loop
```

- Because Cython can use the type information to generate native C code instead of C API calls.

# Technique #2: Direct Indexing

Provide array dimensions for direct indexing:

```
import numpy as np
cimport numpy as cnp
def action():
    cdef cnp.ndarray[cnp.double_t, ndim=2] arr0 = np.empty([1000,1000], dtype='float64')
    arr0.fill(0)
    cdef cnp.ndarray[cnp.double_t, ndim=2] arr1 = np.empty([1000,1000], dtype='float64')
    arr1.fill(1)
    cdef int it = 1
    cdef int jt
    while it < 999:
        jt = 1
        while jt < 999:
            arr0[it, jt] += arr1[it-1, jt-1]
            arr0[it, jt] += arr1[it-1, jt  ]
            arr0[it, jt] += arr1[it-1, jt+1]
            arr0[it, jt] += arr1[it  , jt+1]
            arr0[it, jt] += arr1[it+1, jt+1]
            arr0[it, jt] += arr1[it+1, jt  ]
            arr0[it, jt] += arr1[it+1, jt-1]
            arr0[it, jt] += arr1[it  , jt-1]
            jt += 1
        it += 1
    assert 7968032 == arr0.sum()
```

# Direct Indexing Is Way Faster

- Use NumPy API for indexing: 5.74 sec.

```
> python -m timeit -s 'from example2d import
    action' 'action()'
10 loops, best of 3: 5.74 sec per loop
```

- Direct indexing: 139 msec.

```
> python -m timeit -s 'from example2a import
    action' 'action()'
10 loops, best of 3: 139 msec per loop
```

- Static typing and direct indexing make the code run 41 times faster.
- It's only 139 msec / 51.4 msec = 2.7 times slower than the C version.
  - Good enough for many uses.

# Choose Your Wrapping Tool

- Python is two orders of magnitude slower than C.
- If you want to wrap C++, choose Boost.Python.
  - It thinks around C++. The wrapper should be built from bottom up.
- If you want to wrap C, choose Cython.
  - It thinks around Python. The wrapper should be built from top down.
- When working with NumPy arrays, static typing along with direct indexing makes the number-crunching very fast, just 2 times slower than the C counter part.