Extending Python

With C++, Ctypes and Cffi

Andrey Antukh | www.niwi.be | @niwibe

Boost::Python

A C++ library which enables seamless interoperability between C++ and the Python programming language.

Boost::Python

- Performance comparable to native C API.
- Many python object wrappers and automatic conversion between python and C++
- Easy integrated with native C API
- Can delegate the garbage collection to python or manage a memory yourself using shared_ptr<T> as example.
- Compatible with both: python2 and python3

Boost::Python

- No native Bytes support with python3 (Always translate std::string to python str)
- The documentation must be improved.
- The learning curve is slighly high.
- Not compatible with Pypy.

First "Hello World" method:

```
#include <boost/python.hpp>
#include <iostream>

namespace py = boost::python;

void print_helloworld() {
    std::cout << "Hello World" << std::endl;
}

BOOST_PYTHON_MODULE(example1) {
    py::def("print_helloworld", &::print_helloworld);
}
/*
    >>> import example1
    >>> example1.print_helloworld()
    Hello World
*/
```

First Class

```
#include <boost/python.hpp>
#include <iostream>
#include <string>

namespace py = boost::python;

class Foo {
public:
    Foo(const std::string &name) {
        this->name = name;
    }

    void say_hello() {
        std::cout << "Hello " << this->name << "!" << std::endl;
    }

private:
    std::string name;
};

BOOST_PYTHON_MODULE(example2) {
    py::class_<Foo>("Foo", py::init<std::string>())
        .def("say_hello", &Foo::say_hello);
}
```

First Class usage:

```
>>> import example2
>>> instance = example2.Foo("Andrey")
>>> instance.say_hello()
Hello Andrey!
```

Call python methods from C++

```cpp
#include <boost/python.hpp>

namespace py = boost::python;

int length1(const py::object &obj) {
    return py::call_method<int>(obj.ptr(), "__len__");
}

int length2(const py::object &obj) {
    return py::extract<int>(obj.attr("__len__")());
}

BOOST_PYTHON_MODULE(example3) {
    py::def("length1", &::length1);
    py::def("length2", &::length2);
}

/*
    >>> import example3
    >>> example3.length1([1,2,3])
    3
    >>> example3.length2([1,2,3])
    3
*/
```

CTypes

Is a foreign function library for Python. It can be used to wrap these libraries in pure Python.

CTypes

- Requires low knowledge of c
- Very portable, works well on python, pypy, ironpython, jython.
- Low performance ovehead
- Can not easy track c library api changes.

CTypes: Library load

```
>>> import ctypes
>>> import ctypes.util
>>> pathname = ctypes.util.find_library('mhash')
>>> print(pathname)
libmhash.so.2

>>> lib = ctypes.CDLL(pathname)
>>> lib
<CDLL 'libmhash.so.2', handle 11a90e0 at
7f9e4bfede10>
```

CTypes: Function call

```
>>> lib.mhash_inits
Traceback (most recent call last):
  File "", line 1, in
  File "/usr/lib/python3.3/ctypes/__init__.py", line 366, in
__getattr__
    func = self.__getitem__(name)
  File "/usr/lib/python3.3/ctypes/__init__.py", line 371, in
__getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /usr/lib/libmhash.so.2: undefined symbol:
mhash_inits
>>> lib.mhash_init
<_FuncPtr object at 0x7f9e4c046a10>
>>> lib.mhash_init.argtypes = [ctypes.c_int]
>>> lib.mhash_init.restype = ctypes.c_void_p
>>> ptr = lib.mhash_init(17)
```

CTypes: Pass by reference

```
# sample.c contents:
# void sum(int *i, int y) { *i = *i + y; }
# Compile with: gcc -shared -fPIC -o sample1.so sample1.c

>>> import ctypes
>>> lib = ctypes.CDLL("./sample1.so")
>>> x,y = ctypes.c_int(2), ctypes.c_int(4)
>>> lib.sum(ctypes.pointer(x), y)
546986832
>>> print(x, x.value)
c_int(6) 6
```

CTypes: Real example

py-mhash

https://github.com/niwibe/py-mhash

CTypes: mhashlib/api.py

```python
import ctypes
import ctypes.util

def load_library():
    libpath = ctypes.util.find_library('mhash')
    return ctypes.CDLL(libpath)

MHASH_TIGER160  = 15
MHASH_SHA256    = 17

try:
    lib = load_library()
    lib.mhash_init.argtypes = [ctypes.c_int]
    lib.mhash_init.restype = ctypes.c_void_p
    lib.mhash.argtypes = [ctypes.c_void_p, ctypes.c_char_p, ctypes.c_int]
    lib.mhash_end.argtypes = [ctypes.c_void_p]
    lib.mhash_end.restype = ctypes.c_char_p
    lib.mhash_get_block_size.argtypes = [ctypes.c_int]
    lib.mhash_get_block_size.restype = ctypes.c_int
except (OSError, IOError, AttributeError):
    raise ImportError('mhash shared library not found or incompatible')
```

CTypes: mhashlib/base.py

```
from . import api
import binascii

class sha256(object):
    def __init__(self, data=None):
        self.td = api.lib.mhash_init(api.MHASH_SHA256)
        self._result = None

        if not isinstance(data, bytes):
            raise RuntimeError("data must be bytes instance")
        api.lib.mhash(self.td, data, len(data))

    def hexdigest(self):
        if self._result is not None:
            return binascii.hexlify(self._result)

        size = api.lib.mhash_get_block_size(self._hash_type)
        self._result = api.lib.mhash_end(self.td)
        if len(self._result) > size:
            self._result = self._result[:size]
         return binascii.hexlify(self._result)
```

CTypes: py-mhash usage example

```
>>> import mhashlib
>>> instance = mhashlib.sha256(b"hello\n\n")
>>> instance.hexdigest()
b"
50adea61fa4e77ab111b814716097abfd05f83a207b47eb4529bbd4f54e111e0"
```

CFFI

Foreign Function Interface for Python calling C code.

CFFI

- Works well with cpython and pypy2.0
- Low overhead on c calls.
- 2x-5x performance increment on pypy vs ctypes
- Not have support for all C99.
- Does not support C++
- Requires C knowledge

CFFI: Pass by reference

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

CFFI: Real example

Code based on ctypes version of py-mhash

CFFI: mhashlib/api.py (1/2)

```
from cffi import FFI

ffi = FFI()
c_defs = """
typedef enum __hashid {
        MHASH_SHA256               =  17,
        MHASH_TIGER192             =  7
} hashid;

typedef uint32_t mutils_word32;
typedef uint8_t mutils_word8;
typedef char mutils_boolean;

typedef void (*INIT_FUNC)( void*);
typedef void (*HASH_FUNC)(void*, const void*, int);
typedef void (*FINAL_FUNC)(void*);
typedef void (*DEINIT_FUNC)(void*, unsigned char*);
"""
```

CFFI: mhashlib/api.py (2/2)

```
ffi.cdef(c_defs + """
typedef struct __MHASH_INSTANCE {
        mutils_word32 hmac_key_size;
        mutils_word32 hmac_block;
        mutils_word8 *hmac_key;
        mutils_word8 *state;
        mutils_word32 state_size;
        hashid algorithm_given;
        HASH_FUNC hash_func;
        FINAL_FUNC final_func;
        DEINIT_FUNC deinit_func;
} MHASH_INSTANCE;
typedef MHASH_INSTANCE *MHASH;

MHASH mhash_init(hashid type);
mutils_boolean mhash(MHASH thread,
        const void *plaintext, mutils_word32 size);
mutils_word32 mhash_get_block_size(hashid type);
void *mhash_end(MHASH thread);
""")
lib = ffi.dlopen("mhash")
```

CFFI: mhashlib/base.py

```
import api
import binascii

class sha256(object):
    def __init__(self, data=None):
        self.td = api.lib.mhash_init(17)
        self._result = None
        if not isinstance(data, bytes):
                raise RuntimeError("data must be bytes instance")

        _data = api.ffi.new("char[]", data)
        api.lib.mhash(self.td, _data, len(data))

    def digest(self):
        if self._result is not None:
            return binascii.hexlify(self._result)

        _size = api.lib.mhash_get_block_size(17)
        _result = api.lib.mhash_end(self.td)
        self._result = api.ffi.buffer(_result, _size)
        return binascii.hexlify(self._result)
```