Interfacing with Native code from Python

Why

✔ speeding up hot loops
✔ interfacing with native libraries
✔ improving multi-threaded performance (GIL)
✔ interfacing with other environments
✔ enjoying segfaults

How

✔ Python modules
✔ ctypes
✔ cython
✗ SWIG

```python
def primes(kmax):
    """ A really bad routine to compute kmax primes """
    if kmax > 100000:
        kmax = 100000
    k = 0
    n = 2
    p = [0] * kmax
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
        n = n + 1
    return p
```

```c
#ifndef _LIBPRIME_H_
#define _LIBPRIME_H_

#define PRIME_MAX   10000

/* A library for computing sequences of primes */

typedef struct _ctx ctx_t;

ctx_t   *prime_new(unsigned int len);
void    prime_free(ctx_t *ctx);
void    prime_print(ctx_t *ctx);
/* return new array of prime numbers */
int     *prime_get_data(ctx_t *ctx, int *len);

/* fill array with prime numbers */
void    calculate_primes(int *data, int kmax);
/* return new array of prime numbers */
int     *create_primes(int kmax);

#endif /* _LIBPRIME_H_ */
```

```c
#include <Python.h>

#include "libprime.h"

static PyObject* wrap_primes(PyObject* self, PyObject* args)
{
    unsigned int l, i;
    if (!PyArg_ParseTuple(args, "I", &l))
        return NULL;
    int *data = create_primes(l);
    PyObject *lst = PyList_New(l);
    for (i = 0; i < l; i++)
        PyList_SET_ITEM(lst, i, PyInt_FromLong(data[i]));
    free(data);
    return lst;
}

static PyMethodDef ModuleMethods[] =
{
     {"primes", wrap_primes, METH_VARARGS, "Get a string of variable length"},
     {NULL, NULL, 0, NULL},
};

PyMODINIT_FUNC

initpyprime(void)
{
     (void) Py_InitModule("pyprime", ModuleMethods);
}
```

```
from distutils.core import setup, Extension

setup (name = 'PackageName',
        version = '1.0',
        description = 'This is a demo package',
        ext_modules = [Extension('pyprime', sources = ['modprime.c', 'libprime.c'])])
```

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[100000]
    if kmax > 100000:
        kmax = 100000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
        n = n + 1
    return [p[i] for i in range(kmax)]
```

Interfacing with Object Orientated Libraries

```python
import os.path
import ctypes as ct
import ctypes.util

lib = ct.cdll.LoadLibrary(os.path.abspath("libprime.so"))
lib.prime_get_data.restype = ct.POINTER(ct.c_int)
lib.prime_get_data.argtypes = [ct.c_void_p, ct.POINTER(ct.c_int)]

clib = ct.cdll.LoadLibrary(ctypes.util.find_library("c"))

class Prime:
    def __init__(self, n):
        self._ctx = lib.prime_new(n)

    def _print(self):
        lib.prime_print(self._ctx)

    def get_data(self):
        l = ct.c_int()
        data = lib.prime_get_data(self._ctx, ct.byref(l))
        #note the extra data copy here
        pydata = [data[i] for i in range(l.value)]
        #free the old data using c-library free func
        clib.free(data)
        return pydata
```

```
cimport libc.stdlib

cdef extern from "libprime.h":
    ctypedef struct ctx_t:
        pass
    ctx_t *prime_new(unsigned int len)
    void prime_free(ctx_t *ctx)
    void prime_print(ctx_t *ctx)
    int *prime_get_data(ctx_t *ctx, int *len)

cdef class Prime:
    cdef ctx_t *_ctx
    def __cinit__(self, len):
        self._ctx = prime_new(len)
    def __dealloc__(self):
        prime_free(self._ctx)
    def _print(self):
        prime_print(self._ctx)
    def get_data(self):
        cdef int l
        cdef int *d
        d = prime_get_data(self._ctx, &l)
        pyd = [d[i] for i in range(l)]
        libc.stdlib.free(d)
        return pyd
```

Memory Management and Numpy

```python
import os.path
import numpy as np
import ctypes as ct

lib = ct.cdll.LoadLibrary(os.path.abspath("libprime.so"))
lib.calculate_primes.argtypes = [np.ctypeslib.ndpointer(dtype = np.intc),ct.c_int]
lib.create_primes.restype = ct.POINTER(ct.c_int)
lib.create_primes.argtypes = [ct.c_int]

def primes1(n):
    dest = np.empty(n, dtype=np.intc)
    lib.calculate_primes(dest, n)
    return dest

def primes2(n):
    #as_array() is apparently slower, not in my tests... http://goo.gl/Ia7dB
    data = lib.create_primes(n)
    return np.ctypeslib.as_array(data, shape=(1,n))

def primes3(n):
    data = lib.create_primes(n)
    buf = np.core.multiarray.int_asbuffer(
            ct.addressof(data.contents), n * np.dtype(np.intc).itemsize)
    return np.frombuffer(buf, np.intc)
```

```
cimport numpy as c_np
c_np.import_array()

import numpy as np

cdef extern from "libprime.h":
    int     *create_primes(int kmax)
    void    calculate_primes(int *data, int kmax)

def primes1(int kmax):
    cdef c_np.npy_intp shape[1]
    cdef int* arr_ptr = create_primes(kmax)
    shape[0] = kmax
    ndarray = c_np.PyArray_SimpleNewFromData(1, shape, c_np.NPY_INT, <void*>arr_ptr)
    #numpy owns the memory and will free() it for us. There is an implicit
    #assumption that it was malloc'd, so be wary of changes to mem allocation function
    c_np.PyArray_UpdateFlags(ndarray, ndarray.flags.num | c_np.NPY_OWNDATA)
    return ndarray

def primes2(int kmax):
    cdef c_np.ndarray[c_np.int_t, ndim=1, mode='c'] d
    #ascontiguousarray might incur an extra copy, depending on the alignment
    #and the system. np.zeros is also executed in python, so it might be slower than C
    d = np.ascontiguousarray(np.zeros((kmax,), np.int), dtype=np.int)
    calculate_primes(<int*>d.data, kmax)
    return d
```

Other…

```
import scipy.weave as weave

def ramp(result, size, start, end):
    step = (end-start)/(size-1)
    for i in xrange(size):
        result[i] = start + step*i

def ramp_numeric1(result,start,end):
    code = """
           const int size = Nresult[0];
           const double step = (end-start)/(size-1);
           double val = start;
           for (int i = 0; i < size; i++)
               *result++ = start + step*i;
           """
    weave.inline(code,['result','start','end'],compiler='gcc')
```

```
from rpy2.robjects import r
r('x <- rnorm(100)')  # generate x at R
r('y <- x + rnorm(100,sd=0.5)')  # generate y at R
r('plot(x,y)')  # have R plot them
r('lmout <- lm(y~x)')  # run the regression
r('print(lmout)')  # print from R
loclmout = r('lmout') # download lmout from R to Python
print loclmout  # print locally
```

Writing Nice libraries

- http://davidz25.blogspot.com/2011/07/writing-c-library-intro-conclusion-and.html
- http://0pointer.de/blog/projects/libabc.html

Profiling

- http://packages.python.org/line_profiler/

GIL

- http://wiki.python.org/moin/GlobalInterpreterLock

Numpy

- http://www.scipy.org/PerformancePython
- http://www.scipy.org/Cookbook/Ctypes
- http://rebrained.com/?p=458
- http://technicaldiscovery.blogspot.com/2011/06/speeding-up-python-numpy-cython-and.html
- http://stackoverflow.com/questions/3046305/simple-wrapping-of-c-code-with-cython