

# 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

```
def primes(kmax):  
    """ A really bad routine to compute kmax primes """  
    if kmax > 1000000:  
        kmax = 1000000  
    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
```

```

#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_ */

```

```

#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



```
import os.path
import ctypes as ct

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)]

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
        return [data[i] for i in range(l.value)]
```

```

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)
        return [d[i] for i in range(l)]

```

# Memory Management and Numpy

```
cimport numpy as np
import numpy as np
```

```
#something is not working here...
```

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

```
def primes(int kmax):
    cdef np.npy_intp* shape = [0]
    cdef int* arr_ptr = create_primes(kmax)

    shape[0] = kmax
    return np.PyArray_SimpleNewFromData(1, shape, np.NPY_INT, <void*>arr_ptr)
```

```
def primes2(int kmax):
    cdef np.ndarray[np.int, ndim=1] r = np.zeros((kmax,), dtype=np.int)
    calculate_primes(<int *>r.data, kmax)
    return r
```

```

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.prime_get_data.restype = ct.POINTER(ct.c_int)
lib.prime_get_data.argtypes = [ct.c_void_p, ct.POINTER(ct.c_int)]

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

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))
        buf = np.core.multiarray.int_asbuffer(
            ct.addressof(data.contents), l.value * np.dtype(np.intc).itemsize)
        return np.frombuffer(buf, np.intc)

```

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.robj 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
loc_lmout = r('lmout') # download lmout from R to Python
print loc_lmout # 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/](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>