

Wrapping C Libraries with Cython

Ryan Berthold
EAO

Links

- Cython:

- <http://cython.org/>
- <http://docs.cython.org/en/latest/>
- Apache license: permissive, compiled/linked products don't count as derivative works



- DRAMA:

D R A M A 

- <http://drama.aao.gov.au/html/dramaintro.html>
- Copyright AAO, free for noncommercial use

What is Cython?

- Translates Python to C
- Allows easy creation of C Extensions:
shared libraries imported as modules
- Calls back and forth to C code
- Speeds up execution:
 - Compiled rather than interpreted
 - Static typing
 - Direct function calls

Example

```
def f(x):  
    return 1.0 / (x * x + 1.0)
```

```
def sum_f(N):  
    s = 0.0  
    for i in xrange(N):  
        s += f(i)  
    return s
```

```
#python2 -m timeit -s 'from py_sum_f import sum_f' 'sum_f(1000000)'  
#10 loops, best of 3: 257 msec per loop
```

Cythonizing this code reduces to 154 msec per loop,
just from compiling instead of interpreting.

Static Typing

```
def f(double x):  
    return 1.0 / (x * x + 1.0)
```

```
def sum_f(int N):  
    cdef int i  
    cdef double s = 0.0  
    for i in xrange(N):  
        s += f(i)  
    return s
```

#10 loops, best of 3: 102 msec per loop

Translates to a raw C for() loop, but still has to convert f() arg and return value to/from python float and make a PyObject call.

cdef Functions

```
#cython: embedsignature=True, cdivision=True
```

```
cdef double f(double x):  
    return 1.0 / (x * x + 1.0)
```

```
def sum_f(int N):  
    cdef int i  
    cdef double s = 0.0  
    for i in range(N):  
        s += f(i)  
    return s
```

```
#100 loops, best of 3: 7.22 msec per loop
```

Raw C performance, but note f() no longer available to Python scripts (could have used cpdef instead).

Compiler Directives

embedsignature: nicer docstrings

before:

```
FUNCTIONS  
sum_f(...)
```

after:

```
FUNCTIONS  
sum_f(...)  
sum_f(int N)
```

cddivision: divide-by-zero check

overflowcheck: integer arithmetic

boundscheck: array index limits

wraparound: negative indexing

...many more

Annotated HTML

- `cythonize -a file.pyx`
- `setup(ext_modules=cythonize('*.pyx',
annotate=True))`

Produces highlighted, interactive webpage that shows where Python wrapping might be slowing down your generated C code.

Cython: cy_sum_f_3.pyx - Mozilla Firefox

Cython: cy_sum_f_3.pyx

file:///home/ryanb/pydrama_talk/cy_sum_f_3.html

Generated by Cython 0.23.5

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: [cy_sum_f_3.c](#)

```
01: #cython: embedsignature=True, cdivision=True
02:
+03: cdef double f(double x):
+04:     return 1.0 / (x * x + 1.0)
05:
+06: def sum_f(int N):
07:     cdef int i
+08:     cdef double s = 0.0
+09:     for i in range(N):
+10:         s += f(i)
+11:     return s
    __Pyx_XDECREF(__pyx_r);
    __pyx_t_3 = PyFloat_FromDouble(__pyx_v_s); if (unlikely(!__pyx_t_3)) {__pyx_
    __Pyx_GOTREF(__pyx_t_3);
    __pyx_r = __pyx_t_3;
    __pyx_t_3 = 0;
    goto __pyx_L0;
12:
13: #python2 -m timeit -s 'from cy_sum_f_3 import sum_f' 'sum_f(1000000)'
14: #100 loops, best of 3: 7.22 msec per loop
15:
```

Wrapping C Libraries

- Cython can also use C functions that are defined in external libraries, which makes it suitable as a foreign function interface.
- Use 'cpdef' to automatically generate python wrappers where needed, or use 'cdef' and write your own.
- Custom wrappers and classes used at EAO to provide a more user-friendly interface for the DRAMA library.

What is DRAMA?

- Message-based RPC framework developed at the AAO
- Serialization via SDS (“self-defining data system”) and communication via shared memory (single machine) or TCP (network)
- Written in C, plus Tcl/Tk and Perl interfaces
- Used extensively for telescope and instrument control at JCMT and UKIRT

Interface Changes

- Replace status (error) variables with Exceptions
- Convert SDS to/from Python objects (with numpy for multidim arrays) to avoid get/set API
- Extract all useful message info on user's behalf on action entry or wait() return
- Use standard python logging module; tie logging calls to info/error messages
- Decorators for common patterns, e.g. actions that monitor a single remote parameter

External Functions

```
// f.c
double f(double x) {
    return 1.0 / (x * x + 1.0);
}
```

```
# extern_f.pyx
cdef extern double f(double x)
def f_wrapper(x):
    return f(x)
```

```
gcc -c f.c
cythonize extern_f.pyx
gcc -shared -fPIC -I/usr/include/python2.7 -o extern_f.so
    extern_f.c f.o
```

C Header Files

```
cdef extern from "Python.h":  
    object PyString_FromStringAndSize(char *s, Py_ssize_t len)  
  
cdef extern from "DitsTypes.h":  
  
    enum:  
        DITS_MSG_M_MESSAGE  
        DITS_MSG_M_ERROR  
  
    ctypedef long DitsMsgMaskType  
  
    void * DitsMalloc(int size)  
    void DitsFree(void *ptr)  
  
    ctypedef void(*DitsActionRoutineType)(StatusType *status)
```

Cython does NOT parse .h files – “from <file.h>” basically just inserts a “#include <file.h>” into the generated code. You must explicitly declare any types/functions you need.

cimport: .pxd files

```
# extern_f.pxd
cdef extern double f(double x)
```

```
# extern_f.pyx
from extern_f cimport f
def f_wrapper(x):
    return f(x)
```

```
from libc.stdlib cimport malloc, free
from libc.stdio cimport sprintf
from libc.string cimport memset, memcmp, strlen, strcpy
cimport numpy
```

Cython looks for .pxd files on import/include paths.

Note could have declared f() using 'cpdef', which would have auto-generated a python wrapper.

Casting and Passing

```
cdef object _obj_from_sds(SdsIdType id):  
    # severely abridged version  
    cdef StatusType status = 0  
    cdef void* buf  
    cdef ulong buflen  
    name, code, dims = sds_info(id)  
    SdsPointer(id, &buf, &buflen, &status)  
    sbuf = PyString_FromStringAndSize(<char*>buf, buflen)  
    dtype = _sds_code_to_dtype[code]  
    obj = _numpy.ndarray(shape=dims, dtype=dtype, buffer=sbuf).copy()  
    return obj
```

Basically C notation, but uses <> instead of ().

Structs

```
cdef extern ctypedef struct DitsPathInfoType:
    int MessageBytes
    int MaxMessages
    int ReplyBytes
    int MaxReplies

cdef DitsPathInfoType *path =
    <DitsPathInfoType*>malloc(sizeof(DitsPathInfoType))
print path.MessageBytes
```

Member access always uses dot notation, never '->'.

Callbacks

- Many libraries (including DRAMA) require the user to register callback functions, which the framework then calls in response to events.
- Unlike 'ctypes' module, Cython cannot create C wrappers for Python functions at runtime; need to define intermediate C function.
- Framework must provide the callback with a unique id, like userdata or a function name – otherwise must fall back to ctypes' CFUNCTYPE.

UserData Callbacks

```
ctypedef int(*callback_type)(int arg, void *userdata)
cdef extern void register_callback(callback_type f, void *userdata)

cdef int do_callback(int arg, void *userdata):
    return (<object>userdata)(arg)

def register_python(f):
    register_callback(do_callback, <void*>f)
```

Here we just cast a callable object (function, class) as a void* (same as id(), object's address in memory) and let the framework save it for us.

Named Callbacks

```
actions = {}
```

```
cdef void dispatcher(StatusType *status):  
    try:  
        m = Message()  # init from DitsGetEntInfo() etc.  
        r = actions[m.name](m)  
    except:  
        status[0] = DITS__APP_ERROR
```

```
def register_action(name, f):  
    cdef StatusType status = 0  
    cdef DitsActionDetailsType details  
    if len(name) > DITS_C_NAMELEN:  
        raise ValueError('name too long')  
    memset(&details, 0, sizeof(details))  
    details.obey = dispatcher  
    strcpy(details.name, name)  
    DitsPutActions(1, &details, &status)  
    if status != 0:  
        raise BadStatus(status, "DitsPutActions")  
    actions[name] = f
```

Building

```
#!/usr/bin/env python2
```

```
'''setup.py'''
```

```
from distutils.core import setup  
from Cython.Build import cythonize
```

```
setup(  
    name = "test",  
    ext_modules = cythonize('*.pyx', annotate=True)  
)
```

```
# ./setup.py build_ext --inplace  
# ./setup.py install
```

```

from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

Linux = 'Linux'
ext_modules = [
    Extension("drama.__drama__", ["src/drama.pyx"],
        depends=['setup.py',
                 'src/drama.pxd',
                 'src/ditsaltin.h',
                 'src/ditsmsg.h'],
        include_dirs=['./',
                     '/jac_sw/drama/CurrentRelease/include',
                     '/jac_sw/drama/CurrentRelease/include/os/' + Linux,
                     '/jac_sw/itsroot/install/common/include'],
        library_dirs=['./',
                     '/jac_sw/drama/CurrentRelease/lib/' + Linux,
                     '/jac_sw/itsroot/install/common/lib/' + Linux,
                     '/jac_sw/epics/CurrentRelease/lib/' + Linux],
        libraries=['jit', 'expat', 'tide', 'ca', 'Com', 'git',
                  'dul', 'dits', 'imp', 'sds', 'ers', 'mess', 'm'],
        define_macros=[("unix", None), ("DPOSIX_1", None),
                      ("_GNU_SOURCE", None), ("UNIX", None)],
        extra_compile_args=["-fno-inline-functions-called-once"]
    ])
setup(
    name = 'drama',
    cmdclass = {'build_ext': build_ext},
    packages = ['drama'],
    ext_modules = ext_modules
)

```

Installing

- If you plan to install anywhere other than the site-packages directory, need to tell distutils
- Only accepts command-line args or config files, not setup() options; must set before distutils import
- Scripts need \$PYTHONPATH or sys.path set properly to find your module

```
inst = get_install_base()
pyversion = 'python' + sys.version.split()[0].rpartition('.')[0]
with open('setup.cfg', 'w') as f:
    f.write('[build]\n')
    f.write('build-base=./0.' + Linux + '\n')
    f.write('[install]\n')
    f.write('install-lib=%s/lib/%s/%s\n' % (inst, Linux, pyversion))
    f.write('install-scripts=%s/bin/%s\n' % (inst, Linux))
    f.write('install-data=%s/data\n' % (inst))
    f.write('install-headers=%s/include\n' % (inst))
```

Alternatives

- SWIG
 - Can automatically generate Python wrappers for C/C++ libraries
 - Callback functions may need custom C code and/or .i files
- ctypes
 - Part of the Python standard library
 - Can generate callback wrappers on the fly
 - Potentially less boilerplate
- Many others

Questions?