Cython: A Quick Overview

By Jason Paryani

Presentation will be available at:

github.com/jparyani/cython-presentation

Intro

- “**Cython** is a language that makes writing C extensions for the Python language as easy as **Python** itself.” (www.cython.org)
- This makes it easy to:
  - Call directly into C libraries quickly and easily
  - Speed up slow parts of your code
  - Let’s you mix Python and C, quickly and easily

Caveats

- If you aren't familiar with C/C++, parts of this presentation will go over your head.
- I'm going to make extensive use of micro-benchmarks
  - Only intended to give you a vague idea of the speed differences
  - gcc was used with –O3
  - Don't succumb to pre-mature optimization
  - Check out http://docs.cython.org/src/tutorial/profiling_tutorial.html
- Cython isn't for everything

Syntax

- Pretty similar to Python, with just a few extra things thrown in
- Types
- Structs/Unions/Enums
- Function definitions
- Extension Classes
- Extern blocks
- Lots of other stuff that we'll ignore for now

Types

```
cdef int i, j, k
cdef float f, g[42], *h
```

- cdef followed by a valid type signifies a type a variable defined with a type
- All the basic types from C are there ([unsigned] char, int, long, long long, float, double) and object
- You can typedef your own:

```
ctypedef unsigned long Ulong
```

- Also there's:

```
from libc.stdint cimport uint32_t
```

Casting Types

- Casting is done through the < > operator

```
from libc.stdlib cimport atoi

def parse_int(string):
  return atoi(<char *>string)
```

- Python strings->char * are actually a special case
  - Read the docs

Structs/Unions/Enums

```
cdef struct Grail:
    int age
    float volume
cdef union Food:
    char *spam
    float *eggs
cdef enum CheeseType:
    cheddar, edam, camembert
```

Functions

Python definition:

```
def eggs(unsigned long l, f): ...
```

Cython definition:

```
cdef int eggs(unsigned long l, float f): ...
```

Hybrid definition:

```
cpdef int eggs(unsigned long l, float f): ...
```

Equivalent to a union of the 2 above definitions

Function Call Overhead

Cython

```
cdef _test():
    return 1
def test(int n):
    cdef int i
    for i in range(n):
        _test()
```

Compiled with –O0

timeit test(100)

1000000 loops, best of 3: **1.16 us** per loop

timeit test(1000)

100000 loops, best of 3: **9.83 us** per loop

Python

```
def _test():
    return 1
def test(n):
    for i in range(n):
        _test()
```

timeit test(100)

100000 loops, best of 3: **16.1 us** per loop

timeit test(1000)

10000 loops, best of 3: **155 us** per loop

Tail Call Optimization

Cython

```
cdef int _recurse(int n, int
   count):
  if n <= 0:
    return count
  return _recurse(n-1, count+1)

def recurse(int n):
  return _recurse(n, 0)
```

timeit recurse(100)

10000000 loops, best of 3: **134 ns** per loop

Python

```
def recurse(n, count=0):
  if n <= 0:
    return count return
  recurse(n-1, count+1)
```

timeit recurse(100)

10000 loops, best of 3: **31.3 us** per loop

Extension Classes

```
cdef class Shrubbery:
  cdef int width, height
  def __init__(self, w, h):
    self.width = w
    self.height = h
  def describe(self):
    print "This shrubbery is", self.width, \
  "by", self.height, "cubits.“
```

- Pretty simple, it’s just like defining an extension class in C
- Attribute access is just a struct lookup
- Special __cinit__ and __dealloc__ functions

Extern C headers

```
cdef extern from "zmq.h" nogil:
  # blackbox def for zmq_msg_t
  ctypedef void * zmq_msg_t "zmq_msg_t”
…
  enum: ZMQ_PUB # 1
…
  # send/recv
  int zmq_sendmsg (void *s, zmq_msg_t *msg, int flags)
  int zmq_recvmsg (void *s, zmq_msg_t *msg, int flags)
…
  ctypedef struct zmq_pollitem_t:
    void *socket
    int fd
    short events
    short revents
```

Quick example

Cython

```
cdef extern from "math.h":
    double sin(double )
cpdef double cy_sin(double x):
    return sin(x)
```

timeit cy_sin(3.14)

10000000 loops, best of 3: **116** ns per loop

Python

```
from math import sin
```

timeit sin(3.14)

1000000 loops, best of 3: **154** ns per loop

Quick example 2

```
cdef extern from "math.h":
    double sin(double)
```

- This is only callable from other cython code (since it’s a cdef)
- We need to tell the build step to use libm, in order to link the symbol

```
def cy_sin(double x):
    return sin(x)
```

- This wraps the cdef call, in a python def

```
cpdef double cy_sin2(double x):
    return sin(x)
```

- cy_sin2 is callable efficiently from cython, and from python

Wrapping a C library

- Either wrap them in defs or extension classes

```
cdef extern from "zmq.h" nogil:
    void *zmq_socket (void *context, int type)
    int zmq_sendmsg (void *socket, zmq_msg_t
   *msg, int flags)
```

Becomes:

ZMQ Socket

```
cdef class Socket:
  cdef void *handle
  def __cinit__(self, Context context, int socket_type):
    cdef Py_ssize_t c_handle

    c_handle = context._handle
    self.socket_type = socket_type
    self.handle = zmq_socket(<void *>c_handle, socket_type)

  cpdef object send(self, object data, int flags=0):
    cdef zmq_msg_t msg

    copy_data_to_msg(data, &msg)
    rc = zmq_sendmsg(handle, &data, flags)

    if rc < 0: raise ZMQError()
```

For in range is special

Python

```
cdef int i
for i in range(100):
  pass
```

C

```
CYTHON_UNUSED int __pyx_v_i;
int __pyx_t_1;
for (__pyx_t_1 = 0; __pyx_t_1
  < 100; __pyx_t_1+=1) {
  __pyx_v_i = __pyx_t_1;
}
```

Normal for loop

Cython

```
l = list(range(100))
for i in l:
  pass
```

C

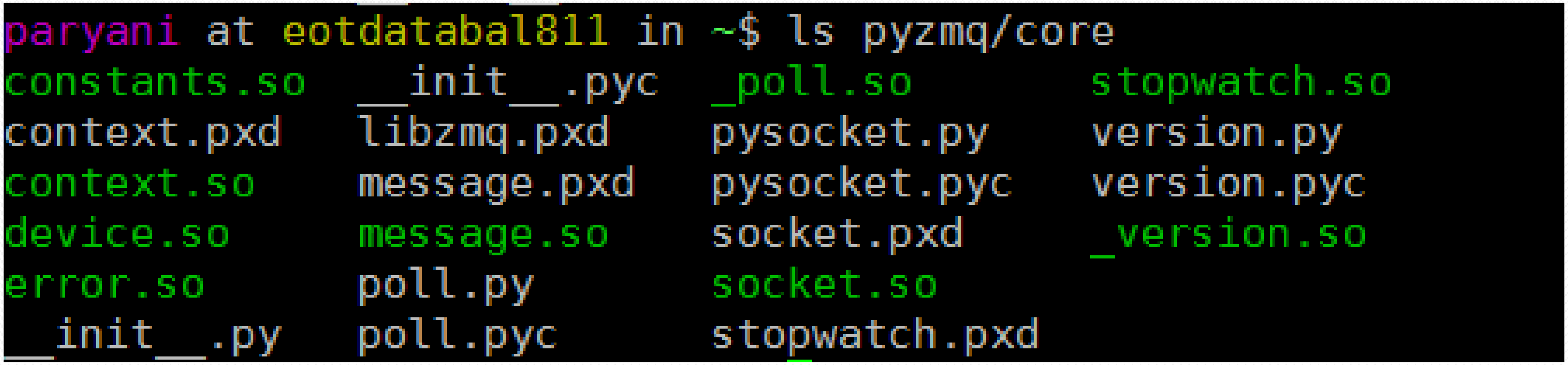
```
__pyx_t_1 = ((PyObject *)__pyx_v_l);
__Pyx_INCREF(__pyx_t_1);
__pyx_t_3 = 0;
for (;;) {
 if (__pyx_t_3 >=
  PyList_GET_SIZE(__pyx_t_1))
    break;
   __pyx_t_2 = PyList_GET_ITEM(__pyx_t_1,
  __pyx_t_3);
   __Pyx_INCREF(__pyx_t_2);
   __pyx_t_3++;
   __Pyx_XDECREF(__pyx_v_i);
   __pyx_v_i = __pyx_t_2;
   __pyx_t_2 = 0;
 }
```

Using Cython interface files

- Cython files come in 3 flavors .pyx, .pxd, and .pxi
- .pyx is where most of your implementation will go
- .pxd is similar to .h files
- For example pyzmq was written almost entirely in Cython, and they export a cython interface alongside a python one

Pyzmq file structure

- Pxd files along with binaries



```
paryani at eotdatabal811 in ~$ ls pyzmq/core
constants.so  __init__.pyc  _poll.so       stopwatch.so
context.pxd   libzmq.pxd    pysocket.py    version.py
context.so    message.pxd   pysocket.pyc   version.pyc
device.so     message.so    socket.pxd     _version.so
error.so      poll.py       socket.so
__init__.py   poll.pyc      stopwatch.pxd
```

Pyzmq

- libzmq.pxd contains a raw wrapping of the zmq C api
- context.pxd and socket.pxd contain definitions of pyzmq's Extension classes, Context and Socket
- These pxd files allow us to import the extension classes in Cython, and access cython specific fields and methods

Cimport

- Usable only from cython
- Very similar to import, but will only import .pxd files in the python path
- Usable like so:

```
from zmq.core.socket cimport Socket
```

Socket.pxd

```
from context cimport Context
cdef class Socket:
  """A 0MQ socket.""“

  cdef void *handle # The C handle for the underlying zmq object.
  cdef public int socket_type # The 0MQ socket type - REQ,REP, etc.
  cdef public Context context # The zmq Context object that owns this.
  cdef public bint _closed # bool property for a closed socket.
  cdef dict _attrs # dict needed for *non-sockopt* get/setattr in
  subclasses
  cdef int _pid # the pid of the process which created me (for fork
  safety)

  # cpdef methods for direct-cython access:
  cpdef object send(self, object data, int flags=*, copy=*, track=*)
  cpdef object recv(self, int flags=*, copy=*, track=*)
```

Simple zmq publisher

```
import zmq

context = zmq.Context()
publisher = context.socket (zmq.PUB)
publisher.bind ("tcp://*:48080")

while True:
    publisher.send('1')
```

Zmq Python client

```
import zmq

context = zmq.Context()
subscriber = context.socket (zmq.SUB)
subscriber.connect("tcp://localhost:48080")
subscriber.setsockopt (zmq.SUBSCRIBE, "")

def main_loop(subscriber):
    count = 0
    while True:
        if count > 1000000: break
    count += int(subscriber.recv())
main_loop(subscriber)
```

C equivalent

```
#include "zmq.h"

int main(int argc, char ** argv) {
  void *context = zmq_init (1);

  // Socket to talk to clients
  void *subscriber = zmq_socket (context, ZMQ_SUB);
  zmq_connect (subscriber, "tcp://localhost:48080");
  zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE, "", 0);
  int count = 0;
  while(1)
  {
    if(count > 1000000) break;
    zmq_msg_t msg;
    zmq_msg_init (&msg);
    zmq_recv (subscriber, &msg, 0);
    count += atoi(zmq_msg_data(&msg));
    zmq_msg_close (&msg);
  }
}
```

Zmq Cython Client

```
# zmq_cython_client.pyx
from zmq.core.socket cimport Socket
from zmq.core.libzmq cimport zmq_msg_t, zmq_msg_init,
  zmq_msg_close, zmq_msg_data, zmq_recvmsg
from libc.stdlib cimport atoi

def main_loop(Socket subscriber):
    cdef int count = 0
    cdef zmq_msg_t msg
    while True:
        if count > 1000000: break
        zmq_msg_init(&msg)
        zmq_recvmsg(subscriber.handle, &msg, 0)
        count += atoi(<char *> zmq_msg_data(&msg))
        zmq_msg_close(&msg)
```

Zmq Python client with Cython

```
import zmq

context = zmq.Context()
subscriber = context.socket (zmq.SUB)
subscriber.connect("tcp://localhost:48080")
subscriber.setsockopt (zmq.SUBSCRIBE, "")

import zmq_cython_client
zmq_cython_client.main_loop(subscriber)
```

Speed comparison

- C ~.33 second per run
- Cython ~.35 second per run
- Python ~1.8 second per run

Building

- Either done through an extension module in a setup.py:

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

ext_modules=[
  Extension(“cython_test", ["cython_test.pyx"])
]
setup(
  name = "cython_test",
  cmdclass = {"build_ext": build_ext},
  ext_modules = ext_modules
)
```

Building 2

- Or through pyximport:

```
import pyximport
pyximport.install()

import cython_test
```

- Read the docs on it, this can get complicated http://docs.cython.org/src/quickstart/build.html

Further Features in Cython

- (Relatively) easy to use GIL management for multithreading
- Also integrates nicely with numpy and the buffer interface
- Fused types (Similar to type templates)
- C++ linking and syntax

Questions?