Cython def, cdef and cpdef functions Documentation

Release 0.1.0

Paul Ross

Sep 07, 2017

Contents

This technical note looks at the different ways of declaring functions in Cython and the performance impact of these choices. We also look at some real world comparions of Python, Cython and C with some surprising results.

CHAPTER 1

Cython Function Declarations

Cython supports three ways of declaring functions using the keywords: `def`, `cdef` and `cpdef`.

`def` - Basically, it's Python

`def` is used for code that will be:

- Called directly from Python code with Python objects as arguments.
- Returns a Python object.

The generated code treats every operation as if it was dealing with Python objects with Python consequences so it incurs a high overhead. `def` is safe to use with no gotchas. Declaring the types of arguments and local types (thus return values) can allow Cython to generate optimised code which speeds up the execution. If the types are declared then a `TypeError` will be raised if the function is passed the wrong types.

`cdef` - Basically, it's C

`cdef` is used for Cython functions that are intended to be pure 'C' functions. All types *must* be declared. Cython aggressively optimises the the code and there are a number of gotchas. The generated code is about as fast as you can get though.

`cdef` declared functions are not visible to Python code that imports the module.

Take some care with `cdef` declared functions; it looks like you are writing Python but actually you are writing C.

`cpdef` - It's Both

`cpdef` functions combine both `def` and `cdef` by creating two functions; a `cdef` for C types and a `def` fr Python types. This exploits early binding so that `cpdef` functions may be as fast as possible when using C fundamental types (by using `cdef`). `cpdef` functions use dynamic binding when passed Python objects and this might much slower, perhaps as slow as `def` declared functions.

CHAPTER 2

How Fast are `def` `cdef` `cpdef`?

Code Example

Here is an example of computing the Fibonacci series (badly) that will be done in Python, Cython and C.

First up, Python [*Fibo.py*]:

```
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)
```

In naive Cython [*cyFibo.pyx*], it is the same code:

```
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)
```

Optimised Cython where we specify the argument type [*cyFibo.pyx*]:

```
def fib_int(int n):
    if n < 2:
        return n
    return fib_int(n-2) + fib_int(n-1)
```

In Cython calling C generated code. Here we use a `def` to call a `cdef` that does the body of the work [*cyFibo.pyx*]:

```
def fib_cdef(int n):
    return fib_in_c(n)

cdef int fib_in_c(int n):
    if n < 2:
        return n
    return fib_in_c(n-2) + fib_in_c(n-1)
```

Now a recursive `cpdef`:

```
cpdef fib_cpdef(int n):
    if n < 2:
        return n
    return fib_cpdef(n-2) + fib_cpdef(n-1)
```

Finally a C extension. We expect this to be the fastest way of computing the result given the algorithm we have chosen:

```
#include "Python.h"

/* This is the function that actually computes the Fibonacci value. */
static long c_fibonacci(long ord) {
    if (ord < 2) {
        return ord;
    }
    return c_fibonacci(ord - 2) + c_fibonacci(ord -1);
}

/* The Python interface to the C code. */
static PyObject *python_fibonacci(PyObject *module, PyObject *arg) {
    PyObject *ret = NULL;
    assert(arg);
    Py_INCREF(arg);
    if (! PyLong_CheckExact(arg)) {
        PyErr_SetString(PyExc_ValueError, "Argument is not an integer.");
        goto except;
    }
    long ordinal = PyLong_AsLong(arg);
    long result = c_fibonacci(ordinal);
    ret = PyLong_FromLong(result);
    assert(! PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    Py_XDECREF(ret);
    ret = NULL;
finally:
    Py_DECREF(arg);
    return ret;
}

/********* The rest is standard Python Extension code ***********/


static PyMethodDef cFiboExt_methods[] = {
{"fib", python_fibonacci, METH_O, "Fibonacci value."},
{NULL, NULL, 0, NULL}           /* sentinel */
};


#if PY_MAJOR_VERSION >= 3

/********* PYTHON 3 Boilerplate ***********/

PyDoc_STRVAR(module_doc, "Fibonacci in C.");

static struct PyModuleDef cFiboExt = {
PyModuleDef_HEAD_INIT,
"cFibo",
module_doc,
-1,
cFiboExt_methods,
NULL,
NULL,
NULL,
NULL
};

PyMODINIT_FUNC
PyInit_cFibo(void)
{
return PyModule_Create(&cFiboExt);
}

#else

/********* PYTHON 2 Boilerplate ***********/


PyMODINIT_FUNC
initcFibo(void)
{
(void) Py_InitModule("cFibo", cFiboExt_methods);
}

#endif
```

Benchmarks

First a correctness check on Fibonacci(30):

```
$ python3 -c "import Fibo, cyFibo, cFibo; print(Fibo.fib(30) == cyFibo.fib(30) == cyFibo.fib_int(30) == cyFibo.fib_cdef(30) == cyFibo.fib_cpdef(30) == cFibo.fib(30))"
True
```

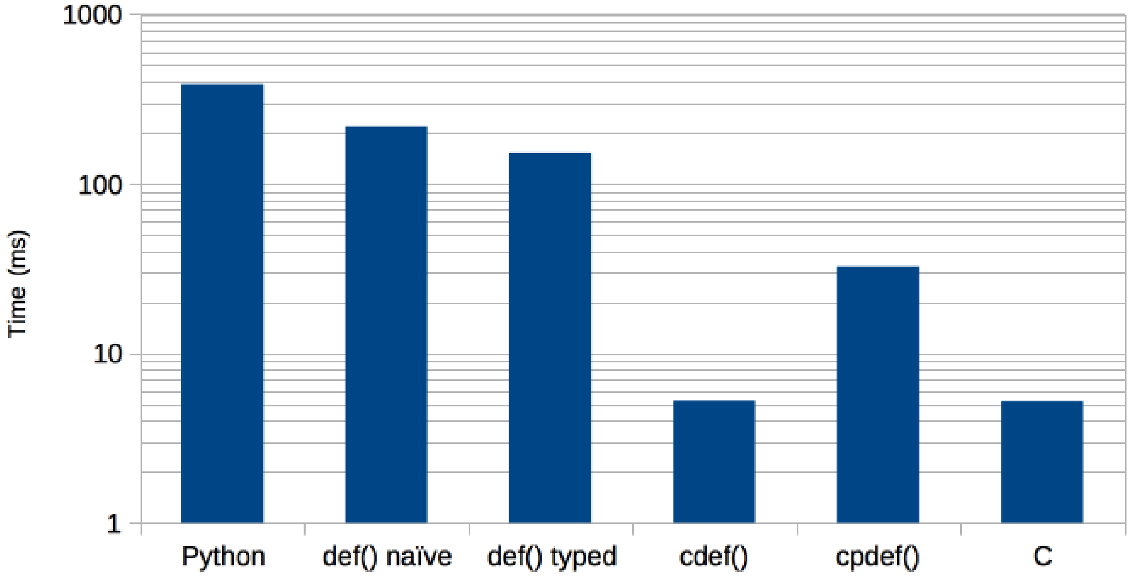
Now time these algorithms on Fibonacci(30) thus:

```
$ python3 -m timeit -s "import Fibo" "Fibo.fib(30)"
$ python3 -m timeit -s "import cyFibo" "cyFibo.fib(30)"
$ python3 -m timeit -s "import cyFibo" "cyFibo.fib_int(30)"
$ python3 -m timeit -s "import cyFibo" "cyFibo.fib_cdef(30)"
$ python3 -m timeit -s "import cyFibo" "cyFibo.fib_cpdef(30)"
$ python3 -m timeit -s "import cFibo" "cFibo.fib(30)"
```

Gives:

Language	Function call	Time (ms)	Speed, Python = 1
Python	`Fibo.fib(30)`	390	x 1
Cython	`cyFibo.fib(30)`	215	x 1.8
Cython	`cyFibo.fib_int(30)`	154	x 2.5
Cython	`cyFibo.fib_cdef(30)`	5.38	x72
Cython	`cyFibo.fib_cpdef(30)`	32.5	x12
C	`cFibo.fib(30)`	5.31	x73

Graphically:



The conclusions that I draw from this are:

- Naive Cython does speed things up, but not by much (x1.8).
- Optimised Cython is fairly effortless (in this case) and worthwhile (x2.5).
- `cdef` is really valuable (x72).
- `cpdef` gives a good improvement over `def` because the recursive case exploits C functions.
- Cython's `cdef` is insignificantly different from the more complicated C extension that is our best attempt.

The Importance of the Algorithm

So far we have looked at pushing code into Cython/C to get a performance gain however there is a glaring error in our code. The algorithm we have been using is **very** inefficient. Here is different algorithm, in pure Python, that will beat all of those above by a huge margin[1]:

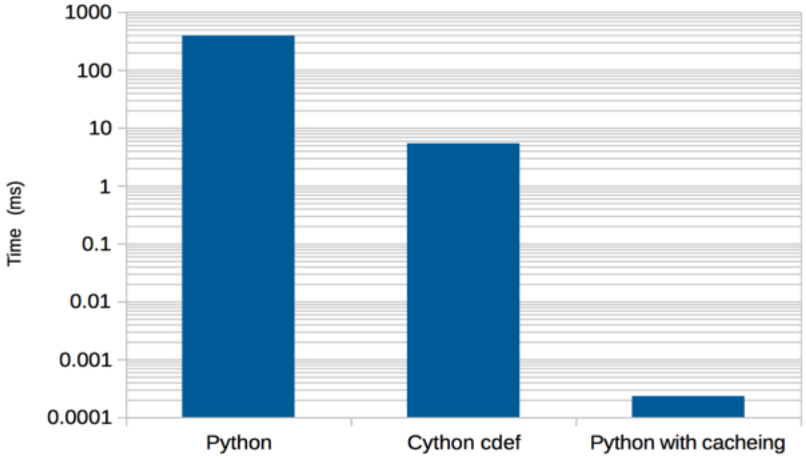
```
def fib_cached(n, cache={}):
    if n < 2:
        return n
    try:
        val = cache[n]
    except KeyError:
        val = fib(n-2) + fib(n-1)
        cache[n] = val
    return val
```

And timing it for Fibonacci(30) gives:

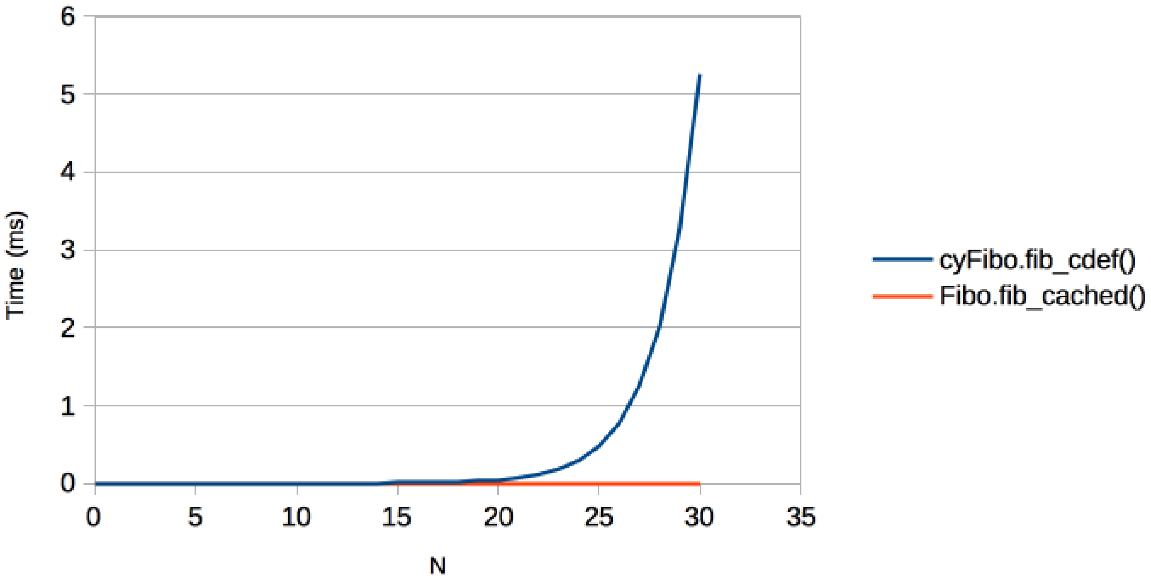
[1] If you are using Python3 you can use the `functools.lru_cache` decorator that gives you more control over cache behaviour.

Language	Function call	Time (ms)	Improvement
Python	`Fibo.fib(30)`	390	x1
Cython	`cyFibo.fib_cdef(30)`	5.38	x72
Python	`Fibo.fib_cached(30)`	0.000231	x1.7e6

Or, graphically:



In fact our new algorithm is far, far better than that. Here is the O(N) behaviour where N is the Fibonacci ordinal:



Hammering a bad algorithm with a fast language is worse than using a good algorithm and a slow language.

CHAPTER 3

Performance of Cython Classes using `def`, `cdef` and `cpdef`

Here we have a class `A` with the three differenct types of method declarations and some means of exercising each one:

```
# File: class_methods.pyx

cdef class A(object):

    def        d(self): return 0
    cdef  int c(self): return 0
    cpdef int p(self): return 0

    def test_def(self, long num):
        while num > 0:
            self.d()
            num -= 1

    def test_cdef(self, long num):
        while num > 0:
            self.c()
            num -= 1

    def test_cpdef(self, long num):
        while num > 0:
            self.p()
            num -= 1
```

We can time the execution of these thus with 1e6 calls:

```
$ python3 -m timeit -s "import cyClassMethods" -s "a = cyClassMethods.A()" "a.test_
↪def(1000000)"
$ python3 -m timeit -s "import cyClassMethods" -s "a = cyClassMethods.A()" "a.test_
↪cdef(1000000)"
$ python3 -m timeit -s "import cyClassMethods" -s "a = cyClassMethods.A()" "a.test_
↪cpdef(1000000)"
```

Call	Result (ms)	Compared to `cdef`
A def	35.9	x17
A cdef	2.15	x1
A cpdef	3.19	x1.5

The Effect of Sub-classing

If we now subclass `A` to `B` where `B` is merely `class B(cyClassMethods.A): pass` and time that:

```
$ python3 -m timeit -s "import cyClassMethods" -s "class B(cyClassMethods.A): pass" -
↪s "b = B()" "b.test_def(1000000)"
$ python3 -m timeit -s "import cyClassMethods" -s "class B(cyClassMethods.A): pass" -
↪s "b = B()" "b.test_cdef(1000000)"
$ python3 -m timeit -s "import cyClassMethods" -s "class B(cyClassMethods.A): pass" -
↪s "b = B()" "b.test_cpdef(1000000)"
```

Call	Result (ms)	Compared to `cdef`
B def	42.6	x20
B cdef	2.17	x1
B cpdef	37.2	x17

We can compare these results with a pure Python implemention:

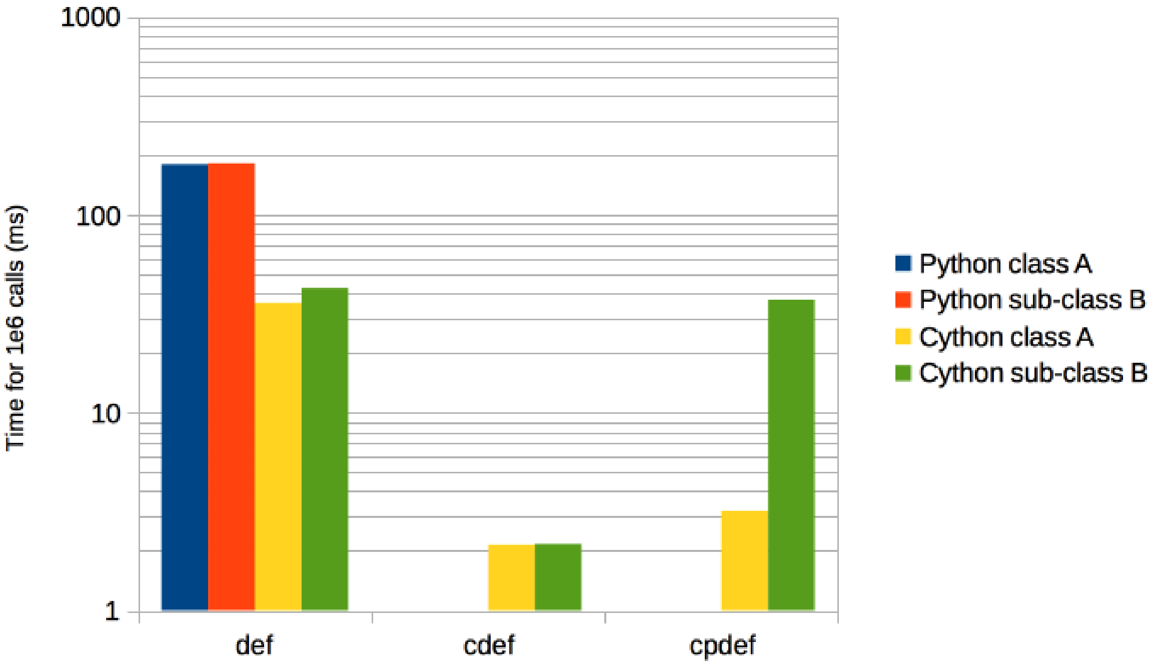
```python
# File: pyClassMethods.py

class PyA(object):

    def d(self): return 0

    def test_d(self, num):
        while num > 0:
            self.d()
            num -= 1

class PyB(A): pass
```

Which we can time with:

```
$ python3 -m timeit -s "import pyClassMethods" -s "a = pyClassMethods.PyA()" "a.test_
↪d(1000000)"
10 loops, best of 3: 180 msec per loop
$ python3 -m timeit -s "import pyClassMethods" -s "b = pyClassMethods.PyB()" "b.test_
↪d(1000000)"
10 loops, best of 3: 182 msec per loop
```

Compared with the Cython `cdef` function these are x84 and x85 respectively.

Graphically the comparison looks like this (note log scale):



My conclusions:

- Cython gives around x4 improvement for normal `def` method calls.
- `cdef` method calls of Cython classes, or those deriving from them, can give a x80 or so performance improvement over pure Python.
- `cpdef` holds up well as a 'safe' `cdef` unless subclassing is used when the cost of the (Python) method lookup brings `cpdef` back to `def` level.

CHAPTER 4

The Performance of Python, Cython and C on a Vector

Lets look at a real world numerical problem, namely computing the standard deviation of a million floats using:

- Pure Python (using a list of values).
- Numpy.
- Cython expecting a numpy array - *naive*
- Cython expecting a numpy array - *optimised*
- C (called from Cython)

The pure Python code looks like this, where the argument is a list of values:

```python
# File: StdDev.py

import math

def pyStdDev(a):
    mean = sum(a) / len(a)
    return math.sqrt((sum(((x - mean)**2 for x in a)) / len(a)))
```

The numpy code works on an ndarray:

```python
# File: StdDev.py

import numpy as np

def npStdDev(a):
    return np.std(a)
```

The naive Cython code also expects an ndarray:

```python
# File: cyStdDev.pyx

import math
```

```
def cyStdDev(a):
    m = a.mean()
    w = a - m
    wSq = w**2
    return math.sqrt(wSq.mean())
```

The optimised Cython code:

```
# File: cyStdDev.pyx

cdef extern from "math.h":
    double sqrt(double m)

from numpy cimport ndarray
cimport numpy as np
cimport cython

@cython.boundscheck(False)
def cyOptStdDev(ndarray[np.float64_t, ndim=1] a not None):
    cdef Py_ssize_t i
    cdef Py_ssize_t n = a.shape[0]
    cdef double m = 0.0
    for i in range(n):
        m += a[i]
    m /= n
    cdef double v = 0.0
    for i in range(n):
        v += (a[i] - m)**2
    return sqrt(v / n)
```

Finally Cython calling pure 'C', here is the Cython code:

```
# File: cyStdDev.pyx

cdef extern from "std_dev.h":
    double std_dev(double *arr, size_t siz)

def cStdDev(ndarray[np.float64_t, ndim=1] a not None):
    return std_dev(<double*> a.data, a.size)
```

And the C code it calls in `std_dev.h`:

```
#include <stdlib.h>
double std_dev(double *arr, size_t siz);
```

And the implementation is in `std_dev.c`:

```
#include <math.h>

#include "std_dev.h"

double std_dev(double *arr, size_t siz) {
    double mean = 0.0;
    double sum_sq;
    double *pVal;
    double diff;
    double ret;
    pVal = arr;
    for (size_t i = 0; i < siz; ++i, ++pVal) {
        mean += *pVal;
    }
    mean /= siz;

    pVal = arr;
    sum_sq = 0.0;
    for (size_t i = 0; i < siz; ++i, ++pVal) {
        diff = *pVal - mean;
        sum_sq += diff * diff;
    }
    return sqrt(sum_sq / siz);
}
```

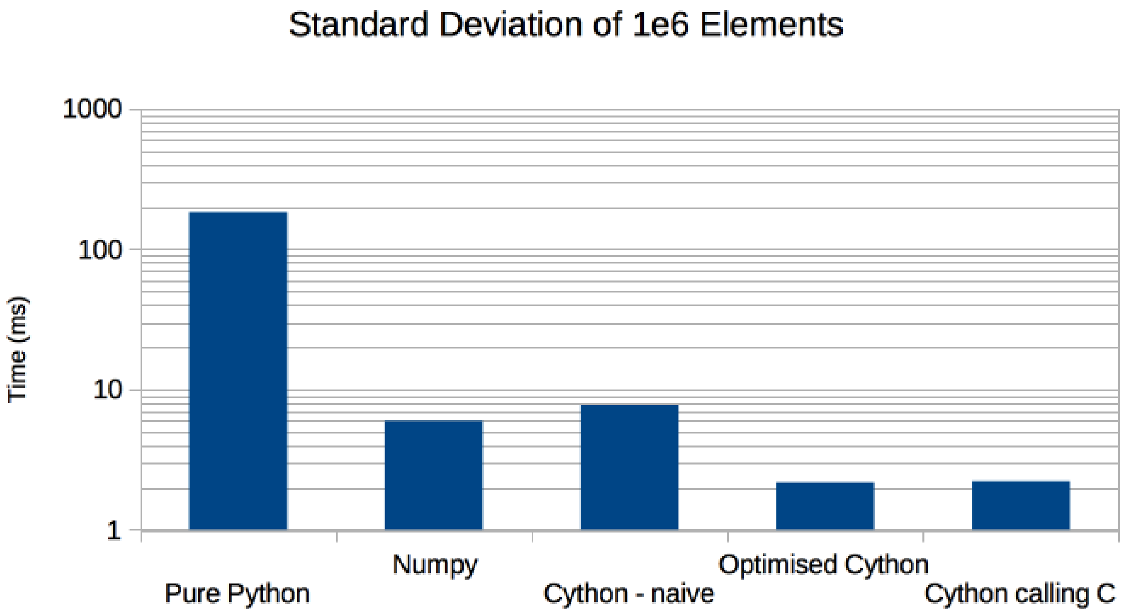
Timing these is done, respectively by:

```
# Pure Python
python3 -m timeit -s "import StdDev; import numpy as np; a = [float(v) for v in␣
↪range(1000000)]" "StdDev.pyStdDev(a)"
# Numpy
python3 -m timeit -s "import StdDev; import numpy as np; a = np.arange(1e6)" "StdDev.
↪npStdDev(a)"
# Cython - naive
python3 -m timeit -s "import cyStdDev; import numpy as np; a = np.arange(1e6)"
↪"cyStdDev.cyStdDev(a)"
# Optimised Cython
python3 -m timeit -s "import cyStdDev; import numpy as np; a = np.arange(1e6)"
↪"cyStdDev.cyOptStdDev(a)"
# Cython calling C
python3 -m timeit -s "import cyStdDev; import numpy as np; a = np.arange(1e6)"
↪"cyStdDev.cStdDev(a)"
```

In summary:

Method	Time (ms)	Compared to Python	Compared to Numpy
Pure Python	183	x1	x0.03
Numpy	5.97	x31	x1
Naive Cython	7.76	x24	x0.8
Optimised Cython	2.18	x84	x2.7
Cython calling C	2.22	x82	x2.7

Or graphically:



The conclusions that I draw from this are:

- Numpy is around 30x faster than pure Python in this case.
- Surprisingly Numpy was not the fastest, even naive Cython can get close to its performance[1].
- Optimised Cython and pure 'C' beat Numpy by a significant margin (x2.7)
- Optimised Cython performs as well as pure 'C' but the Cython code is rather opaque.

[1] At PyconUK 2014 Ian Ozsvald and I may have found why numpy is comparatively slow. Watch this space!

CHAPTER 5

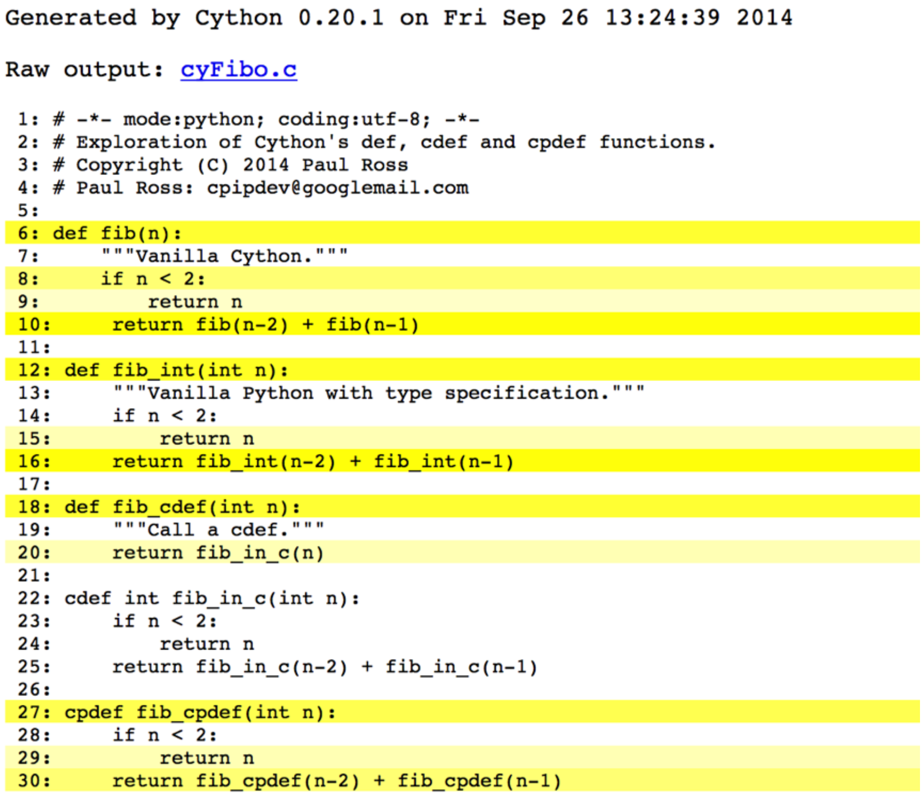
Miscellaneous Notes

Annotating Code With HTML

Cython has a `-a` option which generates an HTML page showing both the Cython code and the C code. It is invoked thus:

```
$ cython -z cyFibo.pyx
```

Here is a screenshot of `the reuslt`. It is colour annotated to show the complexity of the conversion, the more calls into the Python Virtual Machine for each line of Cython then the darker the shade of yellow:

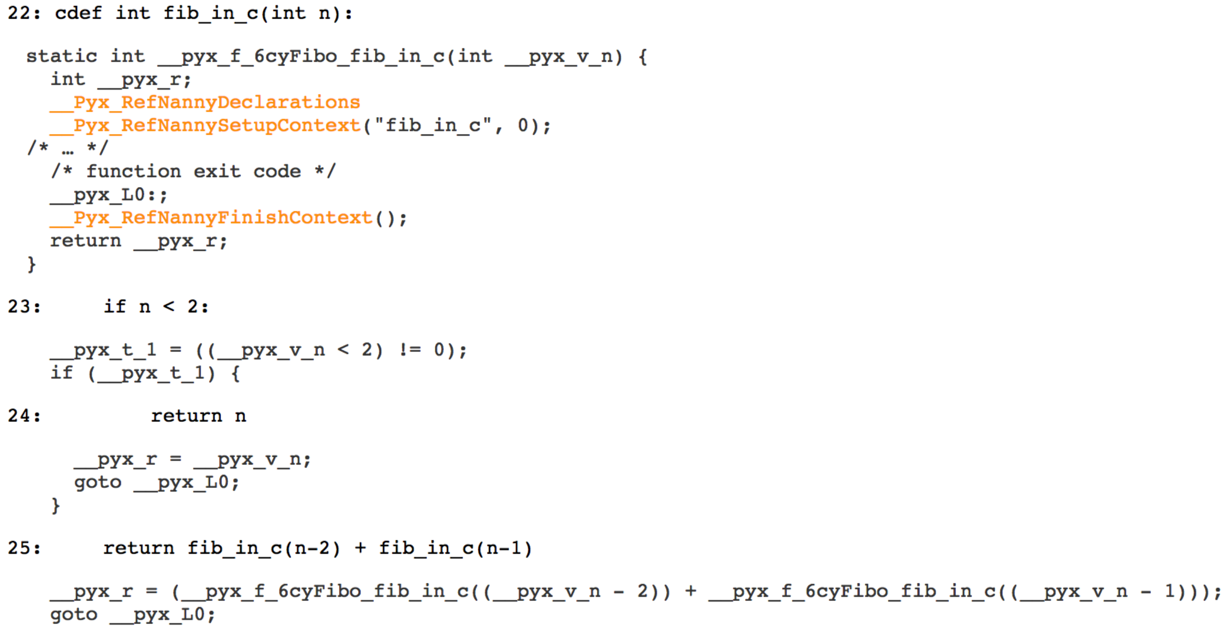


Generated by Cython 0.20.1 on Fri Sep 26 13:24:39 2014

Raw output: cyFibo.c

```
 1: # -*- mode:python; coding:utf-8; -*-
 2: # Exploration of Cython's def, cdef and cpdef functions.
 3: # Copyright (C) 2014 Paul Ross
 4: # Paul Ross: cpipdev@googlemail.com
 5:
 6: def fib(n):
 7:     """Vanilla Cython."""
 8:     if n < 2:
 9:         return n
10:     return fib(n-2) + fib(n-1)
11:
12: def fib_int(int n):
13:     """Vanilla Python with type specification."""
14:     if n < 2:
15:         return n
16:     return fib_int(n-2) + fib_int(n-1)
17:
18: def fib_cdef(int n):
19:     """Call a cdef."""
20:     return fib_in_c(n)
21:
22: cdef int fib_in_c(int n):
23:     if n < 2:
24:         return n
25:     return fib_in_c(n-2) + fib_in_c(n-1)
26:
27: cpdef fib_cpdef(int n):
28:     if n < 2:
29:         return n
30:     return fib_cpdef(n-2) + fib_cpdef(n-1)
```

Clicking on the line numbers expands to reveal the actual C code generated, here all the lines for `fib_in_c()` have been selected:



```
22: cdef int fib_in_c(int n):

  static int __pyx_f_6cyFibo_fib_in_c(int __pyx_v_n) {
    int __pyx_r;
    __Pyx_RefNannyDeclarations
    __Pyx_RefNannySetupContext("fib_in_c", 0);
  /* … */
    /* function exit code */
    __pyx_L0:;
    __Pyx_RefNannyFinishContext();
    return __pyx_r;
  }

23:     if n < 2:

    __pyx_t_1 = ((__pyx_v_n < 2) != 0);
    if (__pyx_t_1) {

24:         return n

      __pyx_r = __pyx_v_n;
      goto __pyx_L0;
    }

25:     return fib_in_c(n-2) + fib_in_c(n-1)

    __pyx_r = (__pyx_f_6cyFibo_fib_in_c((__pyx_v_n - 2)) + __pyx_f_6cyFibo_fib_in_c((__pyx_v_n - 1)));
    goto __pyx_L0;
```

Cython's C code is fairly obfusticated but once you can see The Woman in Red you can call yourself a Cython Ninja.

`cdef`‘ing to a SEGFAULT

Here is an edge case simplified from an issue with Pandas, issue 4519. Suppose we have a number of C functions that return different variations from their argument:

```
/* File: code_value.h */

int code_0(int value) {
    return value + 0;
}

int code_1(int value) {
    return value + 1;
}

int code_2(int value) {
    return value + 2;
}
```

This is then called from this Cython code:

```
# File: code_value.pyx

cdef extern from "code_value.h":
    int code_0(int value)
    int code_1(int value)
    int code_2(int value)

# Create a typedef to a function pointer
ctypedef int (*cv_func)(int value)

# Given an integer code return an appropriate function pointer or raise a ValueError
cdef cv_func get_func(int code):
    if code == 0:
        return &code_0
    elif code == 1:
        return &code_1
    elif code == 2:
        return &code_2
    else:
        raise ValueError('Unrecognised code: %s' % code)

# Get a function pointer then call the function
def code_value(int code, int value):
    cdef:
        cv_func f

    # Get the function pointer
    f = get_func(code)
    # Call the function
    return f(value)
```

This all works fine until we provide a code that is out of range:

```
>>> import cyCodeValue
>>> cyCodeValue.code_value(0, 10)
10
>>> cyCodeValue.code_value(1, 10)
```

```
11
>>> cyCodeValue.code_value(2, 10)
12
>>> cyCodeValue.code_value(3, 10)
Exception ValueError: ValueError('Unrecognised code: 3',) in 'cyCodeValue.get_func'␣
↪ignored
Segmentation fault: 11
```

If we look at the C code that Cython has generated we can see what is going on, I have edited and annotated the code for clarity:

```
    /* "code_value.pyx":16
     *         return &code_2
     *     else:
     *         raise ValueError('Unrecognised code: %s' % code)             # <<<<<<<<
↪<<<<<<
     *
     */

    __pyx_t_2 = __Pyx_PyObject_Call(__pyx_builtin_ValueError, __pyx_t_1, NULL);
    ...
    __pyx_filename = __pyx_f[0];
    __pyx_lineno = 16;
    __pyx_clineno = __LINE__;
    goto __pyx_L1_error;
  }

  /* function exit code */
  __pyx_L1_error:; /* We land here after the ValueError. */
  ...
  __Pyx_WriteUnraisable("cyCodeValue.get_func", __pyx_clineno, __pyx_lineno, __pyx_
↪filename, 0);
  __pyx_r = 0;
  __pyx_L0:;
  __Pyx_RefNannyFinishContext();
  return __pyx_r;
}
```

`get_func()` is declared as a `cdef` that returns a fundamental C type, a function pointer. This suppresses any Python Exception with the call to `__Pyx_WriteUnraisable`, in that case `get_func()` returns 0 which, when dereferenced, causes the SEGFAULT.

`cdef` Exceptions and the Return Type

The Cython documentation says "...a function declared with cdef that does not return a Python object has no way of reporting Python exceptions to its caller. If an exception is detected in such a function, a warning message is printed and the exception is ignored."

Lets see this in isolation:

```
# File: cdef_ret.pyx

def call(val):
    return _cdef(val)

cdef int _cdef(int val):
    raise ValueError('Help')
    return val + 200
```

The `ValueError` will be created, reported, destroyed and the function will return from the *exception* point and the return statement will never be executed. The return value will be the default for the return type, in this case 0:

```
>>> import cyCdefRet
>>> cyCdefRet.call(7)
Exception ValueError: ValueError('Help',) in 'cyCdefRet._cdef' ignored
0
```

The situation changes if we change the declaration of `_cdef()` to `cdef _cdef(int val):` i.e. no declared return type.

In the absence of a return type then Cython assumes a *Python return type* of None so now the exception is **not** ignored and we get:

```
>>> import cyCdefRet
>>> cyCdefRet.call(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "cdef_ret.pyx", line 3, in cyCdefRet.call (cdef_ret.c:718)
    return _cdef(val)
  File "cdef_ret.pyx", line 6, in cyCdefRet._cdef (cdef_ret.c:769)
    raise ValueError('Help')
ValueError: Help
```

If you really want a `cdef` that returns void then declare it as `cdef void _cdef(int val):`

CHAPTER 6

Indices and tables

- genindex
- modindex
- search