Cython: Readable Like Python, Fast Like C

2013-07-27

Zak Fallows

zakf@mit.edu

github.com/zakf/cython_talk

The GitHub repo contains these slides and working code examples.

Presentation License: CC-BY

Talk Outline

1. What is Cython?
2. Why does Cython exist?
3. How does Cython work?
4. How do I use Cython?
5. Appendices

What Is Cython?

Cython is a dialect of Python that allows you to optionally declare a **static type** for each variable.

What Is Cython?

A new programming language

A dialect of Python

A superset of Python*

An alternative implementation of Python**

A Python-to-C code translator***

An optimizing compiler

Talk Outline

1. *What is Cython?*
2. Why does Cython exist?
3. *How does Cython work?*
4. *How do I use Cython?*
5. *Appendices*

Purpose 1: Faster Code

Cython allows Python programmers to write code that is often as fast as hand-written C.

Competing Tools:

PyPy

Numba

Numpy

Purpose 2: Wrap C Code

Cython makes it easy to have Python code interact with C code.

Competing Tools:

- ctypes
- SWIG
- Python/C API
- Boost.Python

Other Purposes

1. Make single-file executables (similar to PyInstaller)
2. Parallel processing (via OpenMP)*
3. Obfuscate code, if you are an evil proprietary software maker
4. Wrap C++ and Fortran code for easy access from Python
5. Wrap interpreted Python code so that C can call it (the reverse of normal)

How Much Faster?

If you compile pure (normal) Python code with Cython, the result is usually 1 to 4 times as fast. “1 time as fast” means “no performance benefit”.

If you add static types, the resulting code is often 10 to 200 times as fast. This is hugely variable, it can be up to 1,000 times or more.

With static types, Cython is generally as fast as hand-written C, but it is easier to read, write, and debug, especially if you love Python.

How Much Faster?

I converted some statistics code from Python to Cython and it became **70x faster.**

I converted some cryptographic hash functions from C# to Cython and they became **4x faster.**

That's right, Cython can be significantly faster than Java and C# in some circumstances.

In your face, Microsoft!

Cython Is Fast in Real Projects

A *k*-dimensional tree is a type of data structure. (You can ask me how they work, but please don't ask me why they are useful.)

The scientific computing package **SciPy** has two modules for manipulating k-d trees:

`scipy.spatial.kdtree` uses normal interpreted Python. This module was written first.

Cython Is Fast in Real Projects

`scipy.spatial.cKDTree` is the same exact code, but it is rewritten in typed Cython. The typed Cython version is often **1,500 times faster** than the pure Python version.

Cython: Readable Like Python, Fast Like C

2013-07-27

Zak Fallows

zakf@mit.edu

github.com/zakf/cython_talk

The GitHub repo contains these slides and working code examples.

Talk Outline

1. *What is Cython?*
2. *Why does Cython exist?*
3. How does Cython work?
4. *How do I use Cython?*
5. *Appendices*

How Cython Works

Cython → C → machine code

How Cython Works

Step 1. **Cython code**: Looks like Python, written by a human. Files: `.pyx, .pxd, .pxi`

Step 2. **C code**: Written by cython.py, hard to read. Files: `.c`

Step 3. **Machine code**: Written by your compiler of choice (GCC, Visual C++, Clang). Files: `.so` (Mac and Linux) or `.pyd` (Windows)

Note: `.pyd` is equivalent to `.dll`

How Cython Works

Cython → C → machine code

`.pyx` → `.c` → `.so`

Terminology

Interpreted Python: Normal Python code running in the normal Python interpreter.

Untyped Cython: Normal Python code which has been compiled to machine code by Cython. This code is dynamically typed. No static types are present. (Small speed boost)

Typed Cython: Some or all variables have static types declared. (Large speed boost)

Terminology

Cython: The subject of this talk.

CPython: This is the standard Python implementation, it is named CPython because it is written in C. CPython is referred to as “the Python interpreter” in this talk.

Do not confuse Cython and CPython, they are different things.

Talk Outline

1. *What is Cython?*
2. *Why does Cython exist?*
3. *How does Cython work?*
4. How do I use Cython?
5. *Appendices*

`ex1.py`: Interpreted Python

```
# ex1.py is short for “example 1”.
# This is a normal Python module (file).

def do_math(seed, power):
    multiplier = 2**(1.0 / power)
    return seed * multiplier

# py> ex1.do_math(3, 4)
# 3.5676213450081633
```

ex1_t.pyx: Typed Cython

```
# The _t in the filename signifies that
# this is “typed Cython”.

cpdef double do_math(long seed, long power):
    cdef double multiplier = 2**(1.0 / power)
    return seed * multiplier

# py> ex1_t.do_math(3, 4)
# 3.567621345008163 3
## Identical result to interpreted Python
```

Cython-specific syntax is in **bold red.**

Syntax: Declaring Variables

To use a normal (dynamic) Python variable, just write normal Python code. No declaration necessary.

To use a **statically typed variable**, which will be higher performance, you must declare it with the **cdef** keyword and the type name.
Examples:

```
cdef int my_num

cdef int my_num2 = 5
```

Notation

A variable name ending in **_py** signifies that it is a Python object, it can change type dynamically.

A variable name ending in **_c** signifies that is a C type, it is statically typed and cannot change type ever.

I will only use this notation sometimes, many identifiers will have no suffix.

Every identifier is either a Python object or a C type, so I could use this notation always.

Statically Typed Variables

Type conversion: Cython will automatically convert Python objects into static variables if possible:

```
num_py = 5                      # A Python object
cdef int num_c = num_py         # Type conversion
cdef float flt_c = num_py # Allowed

flt_py = 7.7
cdef int num2_c = flt_py # Allowed
print num2_c                    # Prints “7”
```

Statically Typed Variables

When using all statically typed variables, you get the bonus of **compile-time type checking.** That is a major reason people love Java and C#. However, when you illegally convert Python objects to statically typed variables, you get runtime errors:

```
str_py = 'hello'
cdef int num3_c = str_py   # Runtime error
# Impossible to convert string to integer
```

Allowable Static Types

All normal C types:

int, long, long long, float, double, char*

C structs, unions, and enums

C function pointers, using normal C syntax

`cdef` classes, which are written like Python classes but are statically typed and are faster

Allowable Static Types

You can also statically declare a variable as a Python type. Example:

```
cdef dict my_dict = {'key': 'val'}
```

This gives you the benefit of **type checking** to avoid type errors. My limited testing found **no performance advantage.**

Allowable Static Types

You can rename static types with the **ctypedef** keyword:

```
ctypedef int8_t int8

ctypedef unsigned long long int ubigint

ctypedef {{ old name }} {{ new name }}
```

Tips for Choosing Types

Integers: Always use the type **long**. The Python interpreter is written in C, and internally it represents Python integers as C long integers. Unless you have a good reason, do not use int or long long.

Note: **long** is usually 32 bits on 32-bit compilers and 64 bits on 64-bit compilers.

Tips for Choosing Types

Warning: Python has two integer types. The first is called “integer” (in Python speak), and it is a C long int. The second is called a “long integer” (in Python speak), and it is an arbitrary precision integer. Python “long integers” print with an L at the end.

Python “long integers” cannot be natively used in Cython typed variables.

Python integer arithmetic never overflows because it will switch from “integer” to “long integer”. Cython arithmetic **will overflow.**

Tips for Choosing Types

Floating points: Always use **double,** because this is used internally by Python for floating point numbers. Unless you have a compelling reason to use float or long double.

Using **long** and **double** will make your Cython code behave like Python code. Using other types can cause different rounding behavior.

Using **double** can actually make your Cython code significantly faster, because it avoids the need to convert the value in memory as it passes between Cython and Python.

Helpful Resource

I do a lot of integer math in Cython, so I made a file that defines useful types:

```
# File: def_types.pxi

from libc.stdint cimport int8_t, uint8_t,
int16_t, uint16_t, int32_t, uint32_t,
int64_t, uint64_t
ctypedef int8_t int8
ctypedef uint8_t uint8
ctypedef int16_t int16
ctypedef uint16_t uint16
ctypedef int32_t int32
ctypedef uint32_t uint32
ctypedef int64_t int64
ctypedef uint64_t uint64
```

Same Example as Before

```
# ex1_t.pyx, Typed Cython

cpdef double do_math(long seed, long power):
    cdef double multiplier = 2**(1.0 / power)
    return seed * multiplier
```

Cython-specific syntax is in **bold red.**

Syntax: Declaring Functions

Just as there are two kinds of variables (dynamic Python variables and static typed variables), there are two kinds of functions:

Python functions are declared normally. These functions can be called by outside Python code.

C functions are declared with special Cython syntax, and they *cannot* be called by outside Python code. They are only available within Cython.

Syntax: Declaring Functions

```
# File: my_math.pyx

# A Python function:
def math_py(seed):
    return seed * 2

# A C function:
cdef long math_c(int seed):
    return seed * 2
```

Cython in Action

The whole point of Cython is that Cython modules, once compiled, can be imported by normal interpreted Python. If we compile the example `my_math.pyx` from the previous slide, we can use it as follows:

```
py> import my_math

py> my_math.math_py(3)
out> 6

py> my_math.math_c(3)
out> AttributeError: The module 'my_math' has
     no attribute 'math_c'.
```

C Functions Are Faster

1. C functions have static types for the arguments and the return value, and that speeds things up.
2. C functions have much lower function call overhead. In fact, C functions will often be **inlined** by the C compiler (e.g. GCC).

How to Compile Cython

1. Make a file named setup.py, here is an example:

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

setup(name = ”Cool Math",
      ext_modules = cythonize(‘cool_math.pyx’))
```

2. Run the following at the command line:

```
python setup.py build_ext --inplace
```

3. Now you can import and use `cool_math`

Example 2: Only Online

I wrote something called Example 2, but I have removed it from the slides. You can see it online with the other code examples.

Example 3: Let's Speed This Up

```
# In file ex3.py:

def inner_ipy(seed, factor):
    intermediate = seed * factor
    return intermediate % 278351

def middle_ipy(seed, n):
    sum = seed + 34
    for iii in range(n):
        sum += inner_ipy(seed, iii)
        seed += 61
        if sum > 94217452:
            sum %= 621943
        if seed > 6129435:
            seed %= 84125
    return sum

# Continued on next slide...
```

Example 3: Interpreted Python

```
# File ex3.py, continued:

def outer_ipy(seed, n0, n1):
    sum = 0
    for iii in range(n0):
        curr_seed = (seed + iii) % 6943
        sum += middle_ipy(curr_seed, n1)
        if sum > 7245103:
            sum %= 22581
    return sum
```

To benchmark, we will run the code above with `seed = 73, n0 = 1,000,` and `n1 = 10,000.`

Example 3: Benchmarks I

```
# Results, as printed by ex3.py:

Interpreted Python:
    Result = 11611
    Time: 3.67643713951 seconds
```

10 million loops in interpreted Python takes **3.7 seconds.**

`ex3_u.pyx`: Untyped Cython

The code for `ex3_u.pyx` is copy-pasted from `ex3.py`. The only difference is that `ex3.py` is run by the Python interpreter, and `ex3_u.pyx` is compiled by Cython into C and then machine code. I call this **untyped Cython**, because it is compiled but it does not use any special Cython syntax.

Example 3: Benchmarks II

```
# Results, as printed by ex3.py:

Interpreted Python:
    Result = 11611
    Time: 3.67643713951 seconds

Untyped Cython:
    Result = 11611
    Time: 1.72477602959 seconds
    Ratio: 2.132
```

Speed boost factor: **2.1**

`ex3_t.pyx`: Typed Cython

We will now make the code progressively faster by adding static types.

For this round, we will add static types to **all local variables.** Here is a snippet of the modified code:

```
def outer_tpy(seed, n0, n1):
    cdef long iii, curr_seed, sum = 0
    for iii in range(n0):
        curr_seed = (seed + iii) % 6943
        sum += middle_tpy(curr_seed, n1)
        if sum > 7245103:
            sum %= 22581
    return sum
```

All incremental changes are in **bold blue**

Example 3: Benchmarks III

```
Interpreted Python:
    Result = 11611
    Time: 3.67643713951 seconds

Untyped Cython:
    Result = 11611
    Time: 1.72477602959 seconds
    Ratio: 2.132

Typed Cython, Python Functions:
    Result = 11611
    Time: 1.40393590927 seconds
    Ratio: 2.619
```

Speed boost of this stage: **1.2**
Cumulative speed boost factor: **2.6**

`ex3_t.pyx`: Add More Types

For this next round, we will add static types to **all function arguments.** Here is a snippet of the modified code:

```
def outer_tpy2(long seed, long n0, long n1):
    cdef long iii, curr_seed, sum = 0
    for iii in range(n0):
        curr_seed = (seed + iii) % 6943
        sum += middle_tpy(curr_seed, n1)
        if sum > 7245103:
            sum %= 22581
    return sum
```

Example 3: Benchmarks IV

```
Interpreted Python:
    Time: 3.67643713951 seconds

Untyped Cython:
    Time: 1.72477602959 seconds
    Ratio: 2.132

Typed Cython, Python Functions:
    Time: 1.40393590927 seconds
    Ratio: 2.619

Typed Cython, Python Functions II:
    Time: 1.02636504173 seconds
    Ratio: 3.582
```

Speed boost of this stage: **1.4**

Cumulative speed boost factor: **3.6**

Python Functions → C Functions

We will now replace the slow Python functions with fast C functions. This is the old code:

```
def inner_tpy2(long seed, long factor):
    # This is still a Python function.
    cdef long intermediate = seed * factor
    return intermediate % 278351
```

This is the new code with a C function:

```
cdef long inner_tc(long seed, long factor):
    cdef long intermediate = seed * factor
    return intermediate % 278351
```

Example 3: Benchmarks V

```
Interpreted Python:
    Time: 3.67643713951 seconds

Untyped Cython:
    Time: 1.72477602959 seconds
    Ratio: 2.132

Typed Cython, Python Functions:
    Time: 1.40393590927 seconds
    Ratio: 2.619

Typed Cython, Python Functions II:
    Time: 1.02636504173 seconds
    Ratio: 3.582

Typed Cython, C Functions:
    Time: 0.0388560295105 seconds
    Ratio: 94.617
```

Speed boost of this stage: **26**
Cumulative speed boost factor: **95**

The `cpdef` Keyword

You don't need to choose between a fast C function or a convenient Python function, you can have the best of both worlds if you use the **cpdef** keyword to declare your function.

```
cpdef long my_math(long seed):
    # Code goes here
```

The `cpdef` Keyword

`cpdef` creates two versions of the same function. The C function is used within Cython, so it is **fast** and it has **type checking** at compile-time. The Python function is used by external Python code that has imported the Cython module.

You cannot `cpdef` a variable, only a function.

Syntax: Exceptions

Cython has optional exception handling. Python functions handle exceptions normally. C functions do not handle exceptions by default, you must use special syntax if you want exception handling:

```
cdef long func(long arg) except? -1:
    raise AnyPythonException("Oh no!")
```

If an exception occurs during this function, the function will immediately return -1. When the calling function sees -1, it will check for an exception and then raise the exception.

Exception Handling Is Cheap

In Cython, exception handling is quite cheap, it does not slow down your code very much. I recommend using it always.

Syntax: Function Pointers

I am including this only because the online documentation completely omits this topic, and it is important.

If you want to pass a function as an argument to another function, you must use a function pointer.

Syntax: Function Pointers

```
cdef long my_fnc(double n) except? -2:
    # Code goes here

cdef void fnc2(long (*op)(double) except? -2) except *:
    # Use the function pointer:
    cdef long result = op(2.5)

# Now call the function:
fnc2(my_fnc)
```

This is the same as C syntax for function pointers, but you include an “except” clause on the end if necessary. The “except” clauses must match.

Boring Function Pointer Details

You **can** use `ctypedef` for function pointers. The following example matches the type of `my_fnc()` from the last slide:

```
ctypedef long (*FncPtr)(double) except? -2
```

You **cannot** pass a `cpdef` function as an argument, only pure C functions are allowed.

You **cannot** have a `cpdef` function accept function pointers as arguments, only pure C functions can accept function pointers.

`cython.py -a`

As you are learning Cython, you will find that `cython.py -a` is very helpful. Use it like this, from the command line:

```
cython.py -a my_module.pyx
```

It will create `my_module.html`, which is an annotated HTML file. If you click on any line of Cython code, it will show you the resulting C code.

`cython.py -a`

In the HTML file, lines are highlighted based on how much they use slow Python functionality. For instance, this line gets highlighted severely, it is a slow point:

```
cdef long result = top / bottom
```

If `top` and `bottom` are both statically typed, then that line should be fully typed and fast. Why is it using slow Python functionality?

Why That Line Is Slow

By default in Cython, a “simple” integer division operation leads to very complex C code. The C code does the following:

1. Check if the two operands have opposite sign.
2. If they do, check whether negating either operand will cause an overflow (very unlikely).
3. Negate as necessary to produce the normal Python-style result. (The C-style result differs if the two operands have opposite signs.)
4. Check if the divisor is zero and raise a ZeroDivisionError if so.

This process requires calls to the `Python.h` library.

How to Fix It

To speed up that code, you must do something like this:

```
cimport cython

@cython.cdivision(True)
cdef long func(long top, long bottom):
    cdef long result = top / bottom
```

That will result in very simple and fast C code, but it will lead to different results if `top` and `bottom` have opposite signs.

Talk Outline

1. *What is Cython?*
2. *Why does Cython exist?*
3. *How does Cython work?*
4. *How do I use Cython?*
5. Appendices

Compiler Directives

By the way, that last slide showed how to set **compiler directives.** There are several useful compiler directives.

One compiler directive turns on **type inference.** If you turn that directive on, then the Cython compiler will automatically give variables static types whenever possible. This results in fast code, but the source code is less cluttered by `cdef` statements.

Compiler Directives

Another compiler directive controls the behavior when an **integer overflows**. Do you want to raise a Python exception, or do you want it to truncate the result like C?

Cython Optimizations

Cython applies two layers of optimization:

1. Cython itself optimizes as it converts Cython code into C code.
2. The C compiler (e.g. GCC) performs optimizations as it generates machine code.

Cython Optimizations

When generating C, Cython performs optimizations such as:

1. Convert `for iii in range(num)` into a Python-free C loop.
2. Convert chained `if… elif… else` statements into fast `switch` statements.

Cython has lots of tricks up its sleeve, including optimizations for dictionaries, lists, strings, and the functions `range()` and `enumerate()`.

Cython Is Buzzword-Compliant

Unlike C, Cython has **garbage collection.**

Unlike C, Cython has **exception handling.**

Cython is suitable for **soft real-time** computing.

Cython has optional **type inference,** just like several fashionable functional programming languages.

Unlike Python, Cython has **compile-time type checking** to catch common bugs.

Cython Is Buzzword-Compliant

Unfortunately, you cannot enjoy the benefits of type inference and compile-time error checking at the same time. If you turn on type inference and then you have a type mismatch, Cython will just declare that variable is a Python object, not a static type like you intended. Then you lose speed.

Cython Is Buzzword-Compliant

Unlike JavaScript, Cython is **relatively strongly typed.** (In JavaScript, `“5” + 6 == “56”`)

Cython is generally **type safe** and **memory safe**, so there are very few segmentation faults compared to C.

At the same time, Cython allows you to twiddle bits at a very low level using tricks like explicit type casting. Cast any variable into a `uint8_t*` array and you can do evil bit hacking.

Cython Is Cross-Platform

I have personally used Cython on Mac OS X, Linux, and Windows, in both 32-bit and 64-bit mode. It probably supports other platforms as well.

Installing Cython on Windows can be tricky. I recommend using MinGW as your compiler on Windows, do not use Visual Studio.

Cython works with Python 2.7 and 3.x.

Cython: A Superset of Python?

Cython 0.19.1 is almost a strict superset of Python. That means if you have valid Python code, you can be 99% sure it will compile and run exactly the same under Cython.

One or two years ago, there were Python features that Cython did not support. They have been added.

I said 99% to be conservative. I am unaware of any exceptions.

Cython: An Alternative Implementation of Python?

This is debatable. Examples of genuine alternative implementations include **PyPy** and **Jython,** which were written from scratch using CPython as inspiration.

Cython is actually built on top of CPython, it does not replace it. The C code generated by Cython must be `#include “Python.h”`.

Cython: A Python-to-C Translator?

This is also debatable. Cython does take in Python code and put out C code. However, the C code must include `Python.h`.

Not Quite a Translator

Consider the following Python code:

```
new_list = my_list[2:5]
```

Cython will “translate” it into the following C code*:

```
PyObject* new_list = NULL;
new_list = PySequence_GetSlice(my_list, 2, 5);
```

The function `PySequence_GetSlice()` is defined in CPython, not Cython.

* The machine-written C code is actually far more obtuse, I have paraphrased it for clarity. For instance, `new_list` is called `__pyx_v_new_list`.

Not Quite a Translator

Whether it is running in the Python interpreter or being compiled by Cython, the list slice operation is being done by the same piece of code. The list slice operation still requires Python objects in memory. A hypothetical “genuine” Python-to-C translator might generate C code like this:

// Original Python: `new_list = my_list[2:5]`

```
for (i=2; i<5; i++)
{
  set_item(new_list, i, get_item(my_list, i));
}
```

Cython: Readable Like Python, Fast Like C

2013-07-27

Zak Fallows

zakf@mit.edu

github.com/zakf/cython_talk

The GitHub repo contains these slides and working code examples.

Text Title

Lorem ipsum dolor sit amet. Falconum teren gibum ipsapracto fid mordor. Tiana betum di factori.

Dolorem falconis tremo dubi qux.

Radino tractis yerot gullot. Tromun fe neader el colunis dractis eurote.

Code Slide

```
def foo(arg):
    bar = 3
    baz = bar + arg
    return baz
```