Cython Tutorial

Dag Sverre Seljebotn
d.s.seljebotn@astro.uio.no

University of Oslo

PyData 2012

Getting set up

Tutorial source files should be on USB stick. Also:
`http://github.com/dagss/cython-pydata12`

Cython relies on:

- A C/C++ compiler (can be a problem on Windows)
- Python development headers
  (Ubuntu: `sudo apt-get install python-dev`)

Using scientific Python distributions (such as EPD) solve both of these.

Getting set up

Tutorial source files should be on USB stick. Also:
`http://github.com/dagss/cython-pydata12`

Cython relies on:

- A C/C++ compiler (can be a problem on Windows)
- Python development headers
  (Ubuntu: `sudo apt-get install python-dev`)

Using scientific Python distributions (such as EPD) solve both of these.

If you distribute software in source code form, your users will need the above too.

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - superset of Python
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - superset of Python
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)
- In addition:
  - Add types for speedups (hundreds of times)
  - Easily use native libraries (C/C++/Fortran) directly

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - superset of Python
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)
- In addition:
  - Add types for speedups (hundreds of times)
  - Easily use native libraries (C/C++/Fortran) directly
- How it works: Cython code is turned into C code which uses the CPython API and runtime.
  - Generated C code can be built and run without Cython installed

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - superset of Python
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)
- In addition:
  - Add types for speedups (hundreds of times)
  - Easily use native libraries (C/C++/Fortran) directly
- How it works: Cython code is turned into C code which uses the CPython API and runtime.
  - Generated C code can be built and run without Cython installed
- Has its share of warts, but works now!

Coding in Cython is like coding in Python and C at the same time!

Usecase 1: Library wrapping

- Cython is a popular choice for writing Python interface modules for C libraries
- Works very well for writing a higher-level Pythonized wrapper
- For 1:1 wrapping other tools might be better suited, depending on the exact usecase

Examples: Pytables, hdf4py, mpi4py, pyzmq, ...

Usecase 2: Performance-critical code

Python	⟷	C/C++/Fortran
High-level		Lower-level
Slow		Fast
No variables typed		All variables typed

- Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python

Usecase 2: Performance-critical code

Python	$\longleftrightarrow$	C/C++/Fortran
High-level		Lower-level
Slow		Fast
No variables typed		All variables typed

- Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python
- Cython: Incremental optimization workflow
  - Optimize, don't re-write
  - Only the pieces you need

Examples: scikits-image, pandas, Sage, ...

Not a usecase: Static type checking

- Cython is (partially) statically typed because it has too, not because it wants to
- You still need to run the program to catch a typo

How Cython works

Cython code must be compiled. This happens in two stages:

1. A `.pyx` file is compiled by Cython to a `.c` file, containing the code of a Python extension module

How Cython works

Cython code must be compiled. This happens in two stages:

1. A `.pyx` file is compiled by Cython to a `.c` file, containing the code of a Python extension module
2. The `.c` file is compiled by a C compiler
   - Special compiler flags dictated by the Python installation must be used

The result is a `.so` file (or `.pyd` on Windows) which can be `import`-ed directly into a Python session.

If you are wrapping a C++ library, one can use C++ instead of C.

Ways to build Cython code...

- `distutils`: `setup.py`

Ways to build Cython code...

- `distutils`: `setup.py`
- Simple cases: `pyximport` automatically compiles on Python import

Ways to build Cython code...

- `distutils`: `setup.py`
- Simple cases: `pyximport` automatically compiles on Python import
- Advanced cases: Use a *real* build tool
  - CMake, waf, SCons, Unix Makefiles, IDE projects...

Exercise 1: Building a Cython module

1. Write some Python code in `example.pyx`
2. Get `https://github.com/dagss/cython-pydata12/hello`
3. To build: `python setup.py build_ext -i`
   - `python setup.py install` installs
4. Then simply `import example`
5. Finally have a look at the generated C code

The problem of getting benchmarks

- CPU frequency scaling

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores
- Interruptions by operating system

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores
- Interruptions by operating system
- Warm-start vs. cold-start

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores
- Interruptions by operating system
- Warm-start vs. cold-start
- What numbers to quote: CPU-time or wall-time?

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores
- Interruptions by operating system
- Warm-start vs. cold-start
- What numbers to quote: CPU-time or wall-time?

Remedies:

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores
- Interruptions by operating system
- Warm-start vs. cold-start
- What numbers to quote: CPU-time or wall-time?

Remedies:

- Disable turbo-mode (BIOS), disable CPU frequency scaling

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores
- Interruptions by operating system
- Warm-start vs. cold-start
- What numbers to quote: CPU-time or wall-time?

Remedies:

- Disable turbo-mode (BIOS), disable CPU frequency scaling
- Outer loop: Repeat benchmark many times, take *minimum*
  - Purpose: Avoid OS interruptions, get a warm-start, help against frequency scaling

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores
- Interruptions by operating system
- Warm-start vs. cold-start
- What numbers to quote: CPU-time or wall-time?

Remedies:

- Disable turbo-mode (BIOS), disable CPU frequency scaling
- Outer loop: Repeat benchmark many times, take *minimum*
  - Purpose: Avoid OS interruptions, get a warm-start, help against frequency scaling
- Inner loop: Repeat benchmark many times, *take average of wall time*
  - Purpose: Get better timer resolution

The problem of getting benchmarks

- CPU frequency scaling
- Turbo-mode when running with one or two cores
- Interruptions by operating system
- Warm-start vs. cold-start
- What numbers to quote: CPU-time or wall-time?

Remedies:

- Disable turbo-mode (BIOS), disable CPU frequency scaling
- Outer loop: Repeat benchmark many times, take *minimum*
  - Purpose: Avoid OS interruptions, get a warm-start, help against frequency scaling
- Inner loop: Repeat benchmark many times, *take average of wall time*
  - Purpose: Get better timer resolution
- Make sure the Cython-part of the benchmark does enough work, or you'll be benchmarking the (slow) speed of a Python for-loop

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef int i, y = two * x
    cdef float z
    z = x / 3
    print z**2
    return z
```

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef int i, y = two * x
    cdef float z
    z = x / 3
    print z**2
    return z
```

- **Types are optional,** and even slightly discouraged. Always check that you have a good reason for using them.

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef int i, y = two * x
    cdef float z
    z = x / 3
    print z**2
    return z
```

- **Types are optional,** and even slightly discouraged. Always check that you have a good reason for using them.
- All C types are available

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef int i, y = two * x
    cdef float z
    z = x / 3
    print z**2
    return z
```

- **Types are optional,** and even slightly discouraged. Always check that you have a good reason for using them.
- All C types are available
- Conversion to and from Python objects happen automatically.

Faster code: Adding types

Variables can be typed with a C-like notation:

```
cdef int two = 2
def func(int x):
    cdef int i, y = two * x
    cdef float z
    z = x / 3
    print z**2
    return z
```

- **Types are optional,** and even slightly discouraged. Always check that you have a good reason for using them.
- All C types are available
- Conversion to and from Python objects happen automatically.
- Note that Python `float` and `int` and C `float` and `int` are different things.

Faster code: cdef functions

Python function calls can be expensive – in Cython doubly so because one might need to convert to and from Python objects to do the call.

Therefore Cython provides a syntax for declaring a Cython-only function:

```
cdef double f(double x): return x**3 + 2*x**2
```

The downside is that the function is not available from Python-space. Using `cpdef` makes a fast version available to Cython and a slower one to Python:

```
cpdef double f(double x): return x**3 + 2*x**2
```

Exercise 2: Adding types

Add types etc. to speed up the following code (name the file `integrate.pyx` for future reference):

```
from __future__ import division
def f(x): return 1/(x**3 + 2*x**2)
def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

- Use the `double` type for function values and domain, and `ssize_t` for integers.
- Expect about 100x speedup. It is easy to miss typing a variable; experiment and see what happens to speed then.
- Use the `-a` switch to the `cython` command-line tool; the generated view of the code explains the speed differences
- Experiment with letting `f` be declared `cdef` vs. `cpdef`.

Raising exceptions from `cdef` functions

For speed reasons, some manual exception declarations must be done on `cdef` functions capable of raising exceptions.

You can always add "`except *`" and not worry about this though.

```
cdef int is_monday() except -2:
    # Must never return -2 explicitly!
    return time.localtime().tm_wday == 0

cdef int divide(int a, int b) except? 45345:
    # If result is 45345, make an additional check
    return a // b # possible ZeroDivisionError

cdef int divide(int a, int b) except *:
    # I'd rather not bother, just ask Python
    return a // b
```

Calling C functions

One *can* do "`from math import sin`" to get Python's `sin` function. Calling C's `sin` function is faster though:

```
cdef extern from "math.h":
    double sin(double)
# or:
# from libc.math cimport sin

cdef double f(double x):
    return sin(x * x)
```

Note that one must "redeclare" the function from `math.h` for the benefit of Cython. In the C compilation stage, only the declaration in `math.h` is seen.

Calling C functions

When calling C functions, one must take care to link in the appropriate libraries. In `setup.py`:

```
...
    Extension("integrate", ["integrate.pyx"],
              libraries=["m"]) # Unix-like specific
...
```

Exercise 3: Call a C function

Just make sure you now know how to calculate $\int_a^b \sin(x^2)\,\mathrm{d}x$.
Check the speed difference between calling Python's `sin` and C's `sin`.

NumPy and Cython

- Cython provides fast access to NumPy arrays
  - As long as datatype and dimensionality of all arrays is known at compile-time!
- 1000x speedup over pure Python loops in extreme cases

NumPy and Cython

- Cython provides fast access to NumPy arrays
  - As long as datatype and dimensionality of all arrays is known at compile-time!
- 1000x speedup over pure Python loops in extreme cases

Some usecases:

- Make manual for-loop-style programming feasible

NumPy and Cython

- Cython provides fast access to NumPy arrays
  - As long as datatype and dimensionality of all arrays is known at compile-time!
- 1000x speedup over pure Python loops in extreme cases

Some usecases:

- Make manual for-loop-style programming feasible
- Optimize an array expression like `1.2 * a + b + np.sqrt(c)`
  - Helps because it reduces memory bandwidth
  - `numexpr` and `Theano` solves the same problem

NumPy and Cython

- Cython provides fast access to NumPy arrays
  - As long as datatype and dimensionality of all arrays is known at compile-time!
- 1000x speedup over pure Python loops in extreme cases

Some usecases:

- Make manual for-loop-style programming feasible
- Optimize an array expression like `1.2 * a + b + np.sqrt(c)`
  - Helps because it reduces memory bandwidth
  - `numexpr` and `Theano` solves the same problem
- Push data between NumPy arrays and C/C++/Fortran libraries

Exercise 4: Optimize pixel-domain convolution

Check out the `integrate` example and speed it up.

- Use the `float` type (32-bit floating point data) for values, `ssize_t` for indices
- NumPy arrays should be typed:

```
import numpy as np
cimport numpy as cnp
cnp.import_array()
def convolve2d(cnp.ndarray[float, ndim=2] f, ...)
```

- Headers needed for the C compiler as well. In `setup.py`:

```
Extension("cy_convolve", ["cy_convolve.pyx"],
          include_dirs=[np.get_include()])
```

Object-oriented programming

Cython has two kinds of classes:

- Normal Python classes. These are exactly like Python's classes.
- Extension types/ "cdef classes"
    - Store typed attributes (avoid converting to/from Python object)
    - Faster method calls when called from Cython
    - Between different Cython modules, `.pxd`-files are needed to export the cdef class interface

```
cdef class MyClass:
    cdef int value
    cpdef int some_method(self, int arg):
        ...
```

Normal Python classes (also in pure Python code) can inherit from cdef classes, but not the other way around.

cdef class polymorphism

Methods of cdef class objects can be called much faster than methods on regular objects, *if* the variable is typed.

```
untyped = MyClass(arg1, arg2)
cdef MyClass typed = untyped
untyped.some_cpdef_method() # slow
typed.some_cpdef_method() # fast
```

Regular Python classes can not be used as the type of a variable.

cdef class attributes

Attributes are different from regular classes:

- All attributes must be pre-declared at compile-time
- Attributes are by default only accessible from Cython (typed access)
- Properties can be declared to make the attribute accessible from Python-space

```
cdef class SineWave(DoubleFunction):
    cdef double offset # not available in Python-space
    cdef public double frequency # available in Python-space
    property period:
        def __get__(self): return 1.0 / self.frequency
        def __set__(self, value): self.frequency = 1.0 / value
    ...
```

cdef class caveats

Special methods have some differences from regular classes:

- `__cinit__` is, unlike `__init__`, guaranteed to be called exactly once per object
    - `__init__` is available as well
    - Take care: Might be called before the Python object has been fully constructed!
    - Rationale: Improper initialization of C attributes can cause leaks or crashes.

cdef class caveats

Special methods have some differences from regular classes:

- `__cinit__` is, unlike `__init__`, guaranteed to be called exactly once per object
    - `__init__` is available as well
    - Take care: Might be called before the Python object has been fully constructed!
    - Rationale: Improper initialization of C attributes can cause leaks or crashes.
- `__dealloc__` is called when the object is deallocated

cdef class caveats

Special methods have some differences from regular classes:

- `__cinit__` is, unlike `__init__`, guaranteed to be called exactly once per object
  - `__init__` is available as well
  - Take care: Might be called before the Python object has been fully constructed!
  - Rationale: Improper initialization of C attributes can cause leaks or crashes.
- `__dealloc__` is called when the object is deallocated
- Arithmetic operators, pickling etc. are different as well

The None issue

- For convenience, variables declared as having a cdef class type can be assigned `None`.
- By default, accessing None in such a "typed" fashion will lead to undefined behaviour (hopefully a crash). Always test with `is None` first!
- The `nonecheck` *compiler directive* will raise an exception instead; but slows down all such accesses.

```
import cython
@cython.nonecheck(True)
def func():
    cdef MyClass obj = None
    try:
        print obj.myfunc() # raises exception
    except AttributeError:
        pass
    with cython.nonecheck(False):
        print obj.myfunc() # hope for a crash!
```

Exercise 5: Using cdef classes in the integration example

Until now, the function to integrate has been hardcoded. Let's use cdef classes to create callbacks without too much of a penalty.