Cython

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

Institute of Theoretical Astrophysics, University of Oslo

Oslo Python Meetup, June 18, 2013

What is Cython?

- Hybrid of Python and C/C++
  - *Almost* a superset of Python
  - Can do almost everything you can in C

What is Cython?

- Hybrid of Python and C/C++
  - *Almost* a superset of Python
  - Can do almost everything you can in C
- Integrated with the CPython runtime
  - `import mycythoncode`

Why?

Bridge Python and C

Why?

Bridge Python and C

- pyzmq, lxml, pymssql, hdf4py, pytables, mpi4py, ...

Why?

Bridge Python and C

- pyzmq, lxml, pymssql, hdf4py, pytables, mpi4py, ...
- Translating idiomatic C/C++ to idiomatic Python
  - (vs. SWIG, ctypes, cffi)

Why?

Speed!

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

Why?

Speed!

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

- For CPU-intensive workloads, Python is slow
  - As in 100x to 1000x slower

Why?

Speed!

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

- For CPU-intensive workloads, Python is slow
  - As in 100x to 1000x slower
- Cython: Incremental optimization workflow
  - Optimize, don't re-write

Why?

Speed!

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

- For CPU-intensive workloads, Python is slow
  - As in 100x to 1000x slower
- Cython: Incremental optimization workflow
  - Optimize, don't re-write
- De facto standard for writing scientific Python libraries
  - Pandas, scikits-learn, parts of SciPy, Sage, tons of domain-specific
  - Often combines number crunching and calling C/Fortran code

Not a usecase: Static type checking

- Cython is (partially) statically typed because it has too, not because it wants to

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 does it work?

1. Write Cython source code (`hello.pyx`)

How does it work?

1. Write Cython source code (`hello.pyx`)
2. Use `cython` to convert to C using the CPython API (`hello.c`)

How does it work?

1. Write Cython source code (`hello.pyx`)
2. Use `cython` to convert to C using the CPython API (`hello.c`)
3. Build using C compiler (`hello.c`) to produce shared library (`hello.so` or `hello.pyd`)

How does it work?

1. Write Cython source code (`hello.pyx`)
2. Use `cython` to convert to C using the CPython API (`hello.c`)
3. Build using C compiler (`hello.c`) to produce shared library (`hello.so` or `hello.pyd`)
4. `import hello`

Ways to build Cython code...

- `distutils`: `setup.py` and `cythonize()`

Ways to build Cython code...

- `distutils`: `setup.py` and `cythonize()`
- IPython notebook

Ways to build Cython code...

- `distutils`: `setup.py` and `cythonize()`
- IPython notebook
- Simple cases: `pyximport` automatically compiles on Python import

Ways to build Cython code...

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

(demo on ways to build)

(demo on speeding up image blur)

(demo on calling libpng)

Using Cython as a language on its own

In addition to Python and C, Cython has its own features:

- Compile-time inclusion of other Cython code using `pxd`-files and `cimport`

Using Cython as a language on its own

In addition to Python and C, Cython has its own features:

- Compile-time inclusion of other Cython code using `pxd`-files and `cimport`
- Avoid overhead of Python function calls with `cdef`-functions

Using Cython as a language on its own

In addition to Python and C, Cython has its own features:

- Compile-time inclusion of other Cython code using `pxd`-files and `cimport`
- Avoid overhead of Python function calls with `cdef`-functions
- Avoid overhead of Python classes with `cdef`-classes

Cython syntax: cdef functions

Calling a `def` function a) has a large performance penalty, can only accept Python-compatible arguments (no pointers).

Syntax for declaring a Cython-only function:

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

Not available from Python-space. Use `cpdef` to make a fast version available to Cython and a slower one to Python:

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

Classes

Cython has two kinds of classes:

- `class MyClass`: Exactly like in Python
- `cdef class MyClass`:
  - Store typed attributes (avoid converting to/from Python object)
  - Faster method calls when called from Cython and declaration known compile-time

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

Normal Python classes 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
- Declare accessability

```
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__` and `__dealloc__` for allocating/freeing C data structures belonging to the object

cdef class caveats

Special methods have some differences from regular classes:

- `__cinit__` and `__dealloc__` for allocating/freeing C data structures belonging to the object
- Arithmetic operators, pickling etc. are different

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!
```

Opinions

- Cython is not elegant but it does many jobs well.

Opinions

- Cython is not elegant but it does many jobs well.
- Cython is a supplement. **Do not** use Cython as your main programming language.

Opinions

- Cython is not elegant but it does many jobs well.
- Cython is a supplement. **Do not** use Cython as your main programming language.
- For speed: Look at Numba (scientific/number crunching) or PyPy (other domains) after making sure they will work for you.