Cython: Compiled Code meets Dynamic Python

Robert Bradshaw
Google, Inc., Seattle

Lisandro Dalcin
Centro Int. de Métodos Computacionales en Ingeniería, Argentina
Python is

- high-level
Python is

- high-level
- interactive
Python is

- high-level
- interactive
- portable
Python is

- high-level
- interactive
- portable
- easy to learn, read, and write
Python is

- high-level
- interactive
- portable
- easy to learn, read, and write
- general purpose
Python is

- high-level
- interactive
- portable
- easy to learn, read, and write
- general purpose
- free and open source
Python is

- high-level
- interactive
- portable
- easy to learn, read, and write
- general purpose
- free and open source

and has large, friendly communities of scientific and non-scientific developers and users.
Python is

- high-level
- interactive
- portable
- easy to learn, read, and write
- general purpose
- free and open source

and has large, friendly communities of scientific and non-scientific developers and users.

...so what’s the catch?
For **some usecases**, Python, *by itself*, is too **slow**.
Performance

What some the options?

- Use a **low-level** language for the **entire** application/stack.
Performance

What some the options?

- Use a **low-level** language for the **entire** application/stack.
  - Does your entire project merit the tedious **write/compile/debug** cycle?
  - "**Premature** optimization is the root of all **evil**."
Performance

What some the options?

- Use a **low-level** language for the **entire** application/stack.
  - Does your entire project merit the tedious **write/compile/debug** cycle?
  - “Premature optimization is the root of all **evil**.”
- Use a **low-level** language with **wrappers**.
Performance

What some the options?

- Use a low-level language for the entire application/stack.
  - Does your entire project merit the tedious write/compile/debug cycle?
  - “Premature optimization is the root of all evil.”
- Use a low-level language with wrappers.
  - Where to draw the line? (see above)
  - Span multiple languages, extra wrapper code.
Performance

What are some options?

- Use *existing, optimized* Python libraries.
Performance

What are some options?

- Use existing, optimized Python libraries.
  - NumPy, SciPy, etc.
  - Heavy lifting done by same BLAS, C, FORTRAN, backends/algorithms as low-level language.
  - Not all algorithms are easily expressed in terms of vectorized rectangular array operations.
What are some options?

- Use **existing, optimized** Python libraries.
  - **NumPy, SciPy**, etc.
  - Heavy lifting done by **same** BLAS, C, FORTRAN, **backends/algorithms** as low-level language.
  - Not all algorithms are **easily expressed** in terms of vectorized **rectangular array** operations.

- Use **Cython**.
What are some options?

- Use **existing, optimized** Python libraries.
  - *NumPy, SciPy*, etc.
  - Heavy lifting done by **same** BLAS, C, FORTRAN, **backends/algorithms** as low-level language.
  - Not all algorithms are **easily expressed** in terms of vectorized **rectangular array** operations.

- Use **Cython**.
  - Spans **high** and **low-level** paradigms.
  - **Complements** existing libraries such as *NumPy*. 
What is Cython?

- Python to C **compiler** (almost)
What is Cython?

- Python to C **compiler** (almost)
- Language extensions for **statically declaring types**
  - Potentially massive speedups
  - Integration with external libraries
What is Cython?

- Python to C **compiler** (almost)
- Language extensions for **statically declaring types**
  - Potentially massive speedups
  - Integration with external libraries
- Python **memory management** and Python object ↔ c data **type conversions** done automatically.
  - removes almost all the headaches of writing C extensions
  - malloc, realloc, free still used for C memory management (could be improved)
What is Cython?

Cython is the missing link between the simplicity of Python and the speed of C / C++ / Fortran.
What is Cython?

Cython is the missing link between the **simplicity of Python** and the **speed of C / C++ / Fortran**.
Why Cython?

- Optimize **only** what you need
Why Cython?

- Optimize **only** what you need
  - Most **time spent** in a **small** amount of code
Why Cython?

- Optimize **only** what you need
  - Most **time spent** in a **small** amount of code
  - Maintain a **consistent codebase**
Why Cython?

- Optimize only what you need
  - Most time spent in a small amount of code
  - Maintain a consistent codebase
- Easy migration
Why Cython?

- Optimize **only** what you need
  - Most **time spent** in a **small** amount of code
  - Maintain a **consistent codebase**
- Easy **migration**
  - **Code** and **developers**
Why Cython?

- Optimize **only** what you need
  - Most **time spent** in a **small** amount of code
  - Maintain a **consistent codebase**
- Easy **migration**
  - **Code** and **developers**
  - Piece by piece
Why Cython?

- Optimize **only** what you need
  - Most **time spent** in a **small** amount of code
  - Maintain a **consistent codebase**
- Easy **migration**
  - **Code** and **developers**
  - Piece by piece
- Leverage **existing libraries**
Why Cython?

- Optimize **only** what you need
  - Most **time spent** in a **small** amount of code
  - Maintain a **consistent codebase**
- Easy **migration**
  - **Code** and **developers**
  - Piece by piece
- Leverage **existing libraries**
  - Directly call **C**, **C++**, and **Fortran** code
Why Cython?

- Optimize **only** what you need
  - Most **time spent** in a **small** amount of code
  - Maintain a **consistent codebase**
- Easy **migration**
  - **Code** and **developers**
  - Piece by piece
- Leverage **existing libraries**
  - Directly call **C**, **C++**, and **Fortran** code
  - Use anything from the **Python ecosystem** as well
How?

Python is **slow** because of...
- its interpreter

Cython is compiled
- dictionary lookups
- complicated calling conventions
- cdef attributes
- cdef functions
- object-oriented primitives
- cdef values and types
How?

Python is **slow** because of...

- its interpreter
- dictionary lookups
Python is **slow** because of...

- its interpreter
- dictionary lookups
- complicated calling conventions
How?

Python is \textbf{slow} because of...

- its interpreter
- dictionary lookups
- complicated calling conventions
- object-oriented primitives
How?

Python is **slow** because of...

- its interpreter
  - *Cython is compiled*
- dictionary lookups
- complicated calling conventions
- object-oriented primitives
Python is **slow** because of...

- its interpreter
  - *Cython is compiled*
- dictionary lookups
  - *Cython has `cdef` attributes*
- complicated calling conventions
- object-oriented primitives
Python is **slow** because of...

- its interpreter
  - *Cython is compiled*
- dictionary lookups
  - *Cython has `cdef` attributes*
- complicated calling conventions
  - *Cython has `cdef` functions*
- object-oriented primitives
Python is **slow** because of...
- its interpreter
  - *Cython is compiled*
- dictionary lookups
  - *Cython has cdef attributes*
- complicated calling conventions
  - *Cython has cdef functions*
- object-oriented primitives
  - *Cython has cdef values and types*
Example

```python
# integrate.py

def f(x):
    return x**2 - x

def integrate_f(a, b, N):
    s = 0.0
    dx = (b - a)/N
    for i in range(N):
        s += f(a + i*dx)
    return s * dx

%timeit integrate_f(0, 1.0, 1000)
625 loops, best of 3: 504 µs per loop
```
Example

_Example_

### integrate.pyx

```python
def f(x):
    return x**2 - x

def integrate_f(a, b, N):
    s = 0.0
    dx = (b - a)/N
    for i in range(N):
        s += f(a + i*dx)
    return s * dx
```

```bash
%timeit integrate_f(0, 1.0, 1000)
625 loops, best of 3: 283 µs per loop
```
Example

integrate.pyx

cdef double f(double x):
    return x**2 - x

def integrate_f(double a, double b, int N):
    cdef int i
    s = 0.0
    dx = (b - a)/N
    for i in range(N):
        s += f(a + i*dx)
    return s * dx

%timeit integrate_f(0, 1.0, 1000)
625 loops, best of 3: 3.77 µs per loop
Cython has **native** support for **NumPy** arrays via the **buffer interface**.

**module.pyx**

```python
cimport numpy as np

# Unpack pointer and stride information
cdef np.ndarray[int, ndim=2] arr = ...

for i in range(arr.shape[0]):
    for j in range(arr.shape[1]):
        ...
        # fast C-level indexing
        arr[i,j] = ...
```
Existing code

What about existing code?

Working, debugged, tested, code has significant advantages over unwritten code.
What about **existing code**?

**Working, debugged, tested**, code has significant advantages over unwritten code.

- Cython’s **static type declarations** make it easy to **call** and link to **existing libraries**.
What about *existing code*?

**Working, debugged, tested**, code has significant advantages over unwritten code.

- Cython’s **static type declarations** make it easy to **call** and link to **existing libraries**.
- Not a wrapper-generator, but easy to write **efficient, Pythonic** wrappers.
What about **existing code**?

**Working, debugged, tested**, code has significant advantages over unwritten code.

- Cython’s **static type declarations** make it easy to **call** and link to **existing libraries**.
- Not a wrapper-generator, but easy to write **efficient, Pythonic** wrappers.
- Ongoing work to to make this even easier (e.g. **Fwrap**).
Declare the functions you need, then use them as if from C.

**gamma.pyx**

```python
cdef extern from "math.h":
    double gamma(double)
    double lgamma(double)

def call_gamma(x):
    return gamma(x), lgamma(x)

>>> call_gamma(3)
(2.0, 0.6931471805599454)
```
Existing code - MPFR

gamma.pxd

cdef extern from "mpfr.h":
    cdef enum mpfr_rnd_t:
        GMP_RNDN

ctypedef void* mpfr_t[1]
cdef int mpfr_init2(mpfr_t, long prec)
cdef int mpfr_init_set_d(
    mpfr_t rop, double op, mpfr_rnd_t rnd)
cdef int mpfr_clear(mpfr_t)
cdef int mpfr_lngamma(
    mpfr_t res, mpfr_t value, mpfr_rnd_t rnd)
cdef int mpfr_sprintf(
    char *buf, char *template, ...)

() Cython SIAM CSE 2011 16 / 21
Existing code - MPFR

gamma.pyx

def log_gamma(double x):
    cdef char[110] buf
    cdef mpfr_t input, res
    mpfr_init_set_d(input, x, GMP_RNDN)
    mpfr_init2(res, 500)
    mpfr_lngamma(res, input, GMP_RNDN)
    mpfr_sprintf(buf, "%.100RNF", res)
    mpfr_clear(input)
    mpfr_clear(res)
    return buf

>>> log_gamma(3)
0.693147180559945309417232121458176568075500134360255254
Who? When?

- **Pyrex** released in 2002 (Greg Ewing), enhanced for **Sage** in 2005-06 (William Stein, Bradshaw), **forked** as Cython in 2007 (Stefan Behnel, Bradshaw). Interest and development pace **grown** at an increasing rate from then. The first Cython **workshop** is in Munich at the end of this month.

- Support from Google (Summer of Code), Enthought, University of Washington, and NSF (via Sage).

- **Hundreds of thousands** of lines of code in a **wide variety** of projects.
Recent developments

There is a huge amount going on

- Type inferance
- C++ support cdef cppclass vector[T]
- gdb support cydgb
- Weave-like functionality cython.inline("...")
- IronPython and .NET backend (ongoing)
- Progress towards full Python compatibility (Cython 1.0 goal, Python regression test suite).

Lots of other exciting directions and optimizations we intend to pursue.
About

Where?

- Web site http://cython.org
- Documentation http://docs.cython.org
- Repository https://github.com/cython/cython
- Bug tracker https://trac.cython.org
- Mailing lists cython-devel@python.org, cython-users@googlegroups.com
Questions?