What is Cython? Python at the speed of C

Python has a reputation for being one of the most convenient, richly outfitted, and downright useful programming languages. Execution speed? Not so much.

Enter Cython. The Cython language is a superset of Python that compiles to C. This yields performance boosts that can range from a few percent to several orders of magnitude, depending on the task at hand. For work bound by Python's native object types, the speedups won't be large. But for numerical operations, or any operations not involving Python's own internals, the gains can be massive.

With Cython, you can skirt many of Python's native limitations or transcend them entirely—without having to give up Python's ease and convenience. In this article, we'll walk through the basic concepts behind Cython and create a simple Python application that uses Cython to accelerate one of its functions.

Compile Python to C

Python code can make calls directly into C modules. Those C modules can be either generic C libraries or libraries built specifically to work with Python. Cython generates the second kind of module: C libraries that talk to Python's internals, and that can be bundled with existing Python code.

Cython code looks a lot like Python code, by design. If you feed the Cython compiler a Python program (Python 2.x and Python 3.x are both supported), Cython will accept it as-is, but none of Cython's native accelerations will come into play. But if you decorate the Python code with type annotations in Cython's special syntax, Cython will be able to substitute fast C equivalents for slow Python objects.

Note that Cython's approach is *incremental*. That means a developer can begin with an *existing* Python application, and speed it up by making spot changes to the code, rather than rewriting the whole application.

This approach dovetails with the nature of software performance issues generally. In most programs, most of the CPU-intensive code is concentrated in a few hot spots—a version of the Pareto principle, also known as the "80/20" rule. Thus, most of the code in a Python application doesn't need to be performance-optimized, just a few critical pieces. You can incrementally translate those hot spots into Cython to get the performance gains you need where it matters most. The rest of the program can remain in Python, with no extra work required.

Advantages of Cython

Aside from being able to speed up the code you've already written, Cython grants several other advantages.

Faster performance working with external C libraries

Python packages like NumPy wrap C libraries in Python interfaces to make them easy to work with. However, going back and forth between Python and C through those wrappers can slow things down. Cython lets you talk to the underlying libraries directly, without Python in the way. (C++ libraries are also supported.)

You can use both C and Python memory management

If you use Python objects, they're memory-managed and garbage-collected the same as in regular Python. If you want to, you can also create and manage your own C-level structures and use `malloc`/`free` to work with them. Just remember to clean up after yourself.

You can opt for safety or speed as needed.

Cython automatically performs runtime checks for common problems that pop up in C, such as out-of-bounds access on an array, by way of decorators and compiler directives (e.g., `@boundscheck (False)`). Consequently, C code generated by Cython is much safer by default than hand-rolled C code, though potentially at the cost of raw performance.

If you're confident you won't need those checks at runtime, you can disable them for additional speed gains, either across an entire module or only on select functions.

Cython also allows you to natively access Python structures that use the buffer protocol for direct access to data stored in memory (without intermediate copying). Cython's memory views let you work with those structures at high speed, and with the level of safety appropriate to the task. For instance, the raw data underlying a Python string can be read in this fashion (fast) without having to go through the Python runtime (slow).

Cython C code can benefit from releasing the GIL

Python's Global Interpreter Lock, or GIL, synchronizes threads within the interpreter, protecting access to Python objects and managing contention for resources. But the GIL has been widely criticized as a stumbling block to a better-performing Python, especially on multicore systems.

If you have a section of code that makes no references to Python objects and performs a long-running operation, you can mark it with the `with nogil:` directive to allow it to run without the GIL. This frees up the Python interpreter to do other things in the interim, and allows Cython code to make use of multiple cores (with additional work).

Cython can be used to obscure sensitive Python code

Python modules are trivially easy to decompile and inspect, but compiled binaries are not. When distributing a Python application to end users, if you want to protect some of its modules from casual snooping, you can do so by compiling them with Cython.

Note, though, that such obfuscation is a *side effect* of Cython's capabilities, not one of its intended functions. Also, it isn't impossible to decompile or reverse-engineer a binary if one is dedicated or determined enough. And, as a general rule, secrets, such as tokens or other sensitive information, should *never* be hidden in binaries—they're often trivially easy to unmask with a simple hex dump.

You can redistribute Cython-compiled modules

If you're building a Python package to be redistributed to others, either internally or via PyPI, Cython-compiled components can be included with it. Those components can be pre-compiled for specific machine architectures, although you'll need to build separate Python wheels for each architecture. Failing that, the user can compile the Cython code as part of the setup process, as long as a C compiler is available on the target machine.

Limitations of Cython

Keep in mind that Cython isn't a magic wand. It doesn't automatically turn every instance of poky Python code into sizzling-fast C code. To make the most of Cython, you must use it wisely—and understand its limitations.

Minimal speedup for conventional Python code

When Cython encounters Python code it can't translate completely into C, it transforms that code into a series of C calls to Python's internals. This amounts to taking Python's interpreter out of the execution loop, which gives code a modest 15 to 20 percent speedup by default. Note that this is a best-case scenario; in some situations, you might see no performance improvement, or even a performance degradation. Measure performance before and after to determine what's changed.

Little speedup for native Python data structures

Python provides a slew of data structures—strings, lists, tuples, dictionaries, and so on. They're hugely convenient for developers, and they come with their own automatic memory management. But they're slower than pure C.

Cython lets you continue to use all of the Python data structures, although without much speedup. This is, again, because Cython simply calls the C APIs in the Python runtime that create and manipulate those objects. Thus Python data structures behave much like Cython-optimized Python code generally: You sometimes get a boost, but only a little. For best results, use C variables and structures. The good news is Cython makes it easy to work with them.

Cython code runs fastest when in 'pure C'

If you have a function in C labeled with the `cdef` keyword, with all of its variables and inline function calls to other things that are pure C, it will run as fast as C can go. But if that function references any Python-native code, like a Python data structure or a call to an internal Python API, that call will be a performance bottleneck.

Fortunately, Cython provides a way to spot these bottlenecks: a source code report that shows at a glance which parts of your Cython app are pure C and which parts interact with Python. The better optimized the app, the less interaction there will be with Python.

Cython and NumPy

Cython improves the use of C-based third-party number-crunching libraries like NumPy. Because Cython code compiles to C, it can interact with those libraries directly, and take Python's bottlenecks out of the loop.

But NumPy, in particular, works well with Cython. Cython has native support for specific constructions in NumPy and provides fast access to NumPy arrays. And the same familiar NumPy syntax you'd use in a conventional Python script can be used in Cython as-is.

However, if you want to create the closest possible bindings between Cython and NumPy, you need to further decorate the code with Cython's custom syntax. The `cimport` statement, for instance, allows Cython code to see C-level constructs in libraries at compile time for the fastest possible bindings.