Learning Cython Programming
Second Edition

Cython is a hybrid programming language used to write C extensions for Python. Combining the practicality of Python and the speed and ease of the C language, Cython is an exciting language that’s worth learning if you want to build fast applications with ease.

This new edition of Learning Cython Programming shows you how to get started, taking you through the fundamentals so you can begin to experience its unique powers.

You’ll find out how to get set up, before exploring the relationship between Python and Cython. You’ll also look at debugging Cython, before moving on to C++ constructs, Python threading, and GIL in Cython. Finally, you’ll learn about object initialization and compile time, and gain a deeper insight into Python 3, which will help you not only become a confident Cython developer, but a much more fluent Python developer too.

Who this book is written for
This book is for developers who are familiar with the basics of C and Python programming and wish to learn Cython programming to extend their applications.

What you will learn from this book
- Reuse Python logging in C
- Make an IRC bot out of your C application
- Extend an application so you have a web server for rest calls
- Practice Cython against your C++ code
- Discover tricks to work with Python ConfigParser in C
- Create Python bindings for native libraries
- Find out about threading and concurrency in relation to GIL
- Expand Terminal Multiplexer Tmux with Cython

Who this book is written for
This book is for developers who are familiar with the basics of C and Python programming and wish to learn Cython programming to extend their applications.


Philip Herron

$ 34.99 US
£ 22.99 UK

Prices do not include local sales tax or VAT

Learn the fundamentals of Cython to extend the legacy of your applications
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Cython Won't Bite'
- A synopsis of the book’s content
**Philip Herron** is a developer who focuses his passion toward compilers and virtual machine implementations. When he was first accepted to Google Summer of Code 2010, he used inspiration from Paul Biggar’s PhD on the optimization of dynamic languages to develop a proof of the concept GCC frontend to compile Python. This project sparked his deep interest in how Python works.

After completing a consecutive year on the same project in 2011, Philip applied to Cython under the Python foundation to gain a deeper appreciation of the standard Python implementation. Through this he started leveraging the advantages of Python to control the logic in systems or even add more high-level interfaces, such as embedding Flask web servers in a REST API to a system-level piece of software, without writing any C code.

Philip currently works as a software consultant for Instil Software based in Northern Ireland. He develops mobile applications with embedded native code for video streaming. Instil has given him a lot of support in becoming a better engineer.

He has written several tutorials for the UK-based Linux Format magazine on Python and loves to share his passion for the Python programming language.
Preface

Cython is a tool that makes writing native extensions to Python as easy as writing them in Python. For those who are unaware, you can implement Python modules as pure the C code, which will, for all intents and purposes, look and act like any Python code. This is required when implementing modules in Python, such as the built-in zip module which use native zlib under the hood. Doing this makes sense for the standard library modules part of Python, though for most users of Python, writing native modules should be the last course of action if possible.

Writing native modules is hard and requires prerequisite knowledge of how to use the garbage collector calls correctly in order to avoid memory leaks. It also requires an understanding of how the GIL is used, which changes if you are using CPython or PyPy. It also requires knowledge of the module structures and argument passing internally to the Python runtime. Therefore, it isn't a trivial process when the need arises. Cython lets us write and work with the native code without having to know anything about the Python runtime. We can write almost pure Python code that just so happens to let us manipulate C/C++ types and classes. We can call back and forth from the native code and into the Python code.

More importantly, Cython removes the complexity and intrinsicity and lets the programmer focus on solving problems.

What this book covers

Chapter 1, Cython Won't Bite, introduces core concepts and demonstrates Cython "Hello World". It discusses the typing and type conversion.

Chapter 2, Understanding Cython, acts as a reference throughout the book. We look at custom C types and function pointers. Using this, we will successfully use Python modules directly from C code.
Chapter 3, Extending Applications, uses everything from the previous chapters to write native Tmux commands using Python instead of C/C++.

Chapter 4, Debugging Cython, uses the cygdb wrapper over gdb to debug Cython code.

Chapter 5, Advanced Cython, introduces how well Cython can work with C++ classes and templates. In general, it also covers caveats in Cython.

Chapter 6, Further Reading, briefly looks at the related projects and interesting sources of new learning.
Cython is much more than a programming language. Its origin can be traced to SAGE, the mathematics software package, where it is used to increase the performance of mathematical computations such as those involving matrices. More generally, I tend to consider Cython as an alternative to SWIG to generate really good Python bindings to native code.

Language bindings have been around for years, and SWIG was one of the first and best tools to generate bindings for multitudes of languages. Cython generates bindings for Python code only, and this single purpose approach means it generates the best Python bindings you can get outside of doing it all manually, which should be attempted only if you're a Python core developer.

For me, taking control of legacy software by generating language bindings is a great way to reuse any software package. Consider a legacy application written in C/C++. Adding advanced modern features such as a web server for a dashboard or message bus is not a trivial thing to do. More importantly, Python comes with thousands of packages that have been developed, tested, and used by people for a long time that can do exactly that. Wouldn't it be great to take advantage of all of this code? With Cython, we can do exactly this, and I will demonstrate approaches with plenty of example codes along the way.

This first chapter will be dedicated to the core concepts on using Cython, including compilation, and should provide a solid reference and introduction for all the Cython core concepts.

In this first chapter, we will cover:

- Installing Cython
- Getting started - Hello World
- Using distutils with Cython
Cython Won't Bite

- Calling C functions from Python
- Type conversion

Installing Cython
Since Cython is a programming language, we must install its respective compiler, which just so happens to be the aptly named Cython.

There are many different ways to install Cython. The preferred one would be to use pip:

   $ pip install Cython

This should work on both Linux and Mac. Alternatively, you can use your Linux distribution's package manager to install Cython:

   $ yum install cython     # will work on Fedora and Centos
   $ apt-get install cython # will work on Debian based systems.

For Windows, although there are a plethora of options available, following this wiki is the safest option to stay up-to-date: http://wiki.cython.org/InstallingOnWindows.

Emacs mode
There is an emacs mode available for Cython. Although the syntax is nearly the same as Python, there are differences that conflict in simply using Python-mode. You can grab cython-mode.el from the Cython source code (inside the Tools directory.) The preferred way of installing packages to emacs would be to use a package repository like MELPA:

To add the package repository to emacs, open your ~/.emacs configuration file and add:

   (when (>= emacs-major-version 24)
      (require 'package)
      (add-to-list
       'package-archives
       '("melpa" . http://melpa.org/packages/))
   )
   (package-initialize)
Once you add this and reload your configuration to install the Cython mode, you can simply run:

'M-x package-install RET cython-mode'

Once this is installed, you can activate the mode by adding this into your emacs config file:

(require 'cython-mode)

You can activate the mode manually at any time with:

'M-x cython-mode RET'

**Getting the code examples**

Throughout this book, I intend to show real examples that are easy to digest in order to help you get a feel of the different things you can achieve with Cython. To access and download the code used, please clone this repository:

$ git clone git://github.com/redbrain/cython-book.git

**Getting started – Hello World**

As you will see when running the **Hello World** program, Cython generates native Python modules. Therefore, running any Cython code, you will reference it via a module import in Python. Let's build the module:

$ cd cython-book/chapter1/helloworld
$ make

You should now have created helloworld.so! This is a Cython module of the same name as the Cython source code file. While in the same directory of the shared object module, you can invoke this code by running a respective Python import:

$ python
Python 2.7.3 (default, Aug 1 2012, 05:16:07)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import helloworld
Hello World from cython!
As you can see by opening `helloworld.pyx`, it looks just like a normal Python Hello World application, but as previously stated, Cython generates modules. These modules need a name so that they can be correctly imported by the Python runtime. The Cython compiler simply uses the name of the source code file. It then requires us to compile this to the same shared object name.

Overall, Cython source code files have the `.pyx`, `.pxd`, and `.pxi` extensions. For now, all we care about are the `.pyx` files; the others are for `cimports` and `includes` respectively within a `.pyx` module file.

The following screenshot depicts the compilation flow required to have a callable native Python module:

I wrote a basic makefile so that you can simply run `make` to compile these examples. Here's the code to do this manually:

```bash
$ cython helloworld.pyx
$ gcc/clang -g -O2 -fpic `python-config --cflags` -c helloworld.c -o helloworld.o
$ gcc/clang -shared -o helloworld.so helloworld.o `python-config --libs`
```
Using distutils with Cython

You can also compile this HelloWorld example module using Python distutils and cythonize. Open the setup.py along side the Makefile and you can see the alternate way to compile Cython modules:

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

setup(
    ext_modules = cythonize("helloworld.pyx")
)
```

Using the cythonize function as part of the ext_modules section will build any specified Cython source into an installable Python module. This will compile helloworld.pyx into the same shared library. This provides the Python practice to distribute native modules as part of distutils.

Calling C functions from Python

We should be careful for clarity when talking about Python and Cython since the syntax is so similar. Let's wrap a simple AddFunction in C and make it callable from Python.

First, open a file called AddFunction.c, and write a simple function in it:

```c
#include <stdio.h>

int AddFunction(int a, int b) {
    printf("look we are within your c code!\n");
    return a + b;
}
```

This is the C code that we will call—just a simple function to add two integers. Now, let's get Python to call it. Open a file called AddFunction.h, wherein we will declare our prototype:

```c
#ifndef __ADDFUNCTION_H__
#define __ADDFUNCTION_H__

extern int AddFunction (int, int);

#endif //__ADDFUNCTION_H__
```

[5]
Cython Won’t Bite

We need this so that Cython can see the prototype for the function we want to call. In practice, you will already have your headers in your own project with your prototypes and declarations already available.

Open a file called AddFunction.pyx, and insert the following code in it:

```python
cdef extern from "AddFunction.h":
    cdef int AddFunction(int, int)
```

Here, we have to declare which code we want to call. The `cdef` is a keyword signifying that this is from the C code that will be linked in. Now, we need a Python entry point:

```python
def Add(a, b):
    return AddFunction(a, b)
```

This `Add` function is a Python callable inside a PyAddFunction module this acts as a wrapper for Python code to be able to call directly into the C code. Again, I have provided a handy makefile to produce the module:

```
$ cd cython-book/chapter1/ownmodule
$ make
```

Notice that `AddFunction.c` is compiled into the same `PyAddFunction.so` shared object. Now, let’s call this `AddFunction` and check to see if C can add numbers correctly:

```
$ python
>>> from PyAddFunction import Add
>>> Add(1,2)
look we are within your c code!!
3
```

Notice that the print statement inside the `AddFunction` and the final result are printed correctly. Therefore, we know that the control hit the C code and did the calculation in C, and not inside the Python runtime. This is a revelation of what is possible. Python can be cited to be slow in some circumstances. Using this technique makes it possible for Python code to bypass its own runtime and to run in an unsafe context, which is unrestricted by the Python runtime which is much faster.
Type conversion in Cython

Notice that we had to declare a prototype inside the Cython source code `PyAddFunction.pyx`:

```python
cdef extern from "AddFunction.h":
    cdef int AddFunction(int, int)
```

It lets the compiler know that there is a function called `AddFunction` and it takes two ints and returns an int. This is all the information the compiler needs to know beside the host and target operating system's calling convention to call this function safely. Then, we created the Python entry point, which is a Python callable that takes two parameters:

```python
def Add(a, b):
    return AddFunction(a, b)
```

Inside this entry point, it simply returned the native `AddFunction` and passed the two Python objects as parameters. This is what makes Cython so powerful. Here, the Cython compiler must inspect the function call and generate code to safely try and convert these Python objects to native C integers. This becomes difficult when precision is taken into account as well as potential overflow, which just so happens to be a major use case since it handles everything so well. Also, remember that this function returns an integer, and Cython also generates code to convert the integer return into a valid Python object.

---

Summary

Overall, we installed the Cython compiler, ran the Hello World example, and took into consideration that we need to compile all code into native shared objects. We also saw how to wrap native C code to make it callable from Python. We have also seen the implicit type conversion which Cython does for us to make calling C work. In the next chapter, we will delve deeper into Cython programming with discussion on how to make Python code callable from C and manipulate native C data structures from Cython.

---
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.