



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Learning Cython Programming

Expand your existing legacy applications in C using Python

Philip Herron

[PACKT] open source*
PUBLISHING community experience distilled

Learning Cython Programming

Expand your existing legacy applications in C
using Python

Philip Herron

[PACKT] open source 
PUBLISHING community experience distilled
BIRMINGHAM - MUMBAI

Learning Cython Programming

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2013

Production Reference: 1190913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-079-7

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author

Philip Herron

Project Coordinator

Romal Karani

Reviewers

Namit Kewat

Goran Milovanovic

Proofreader

Paul Hindle

Acquisition Editor

Antony Lowe

Indexers

Mehreen Deshmukh

Rekha Nair

Commissioning Editor

Mohammed Fahad

Graphics

Sheetal Atule

Technical Editor

Hardik B. Soni

Production Coordinator

Kirtee Shingan

Copy Editors

Sayanee Mukherjee

Aditya Nair

Kirti Pai

Adithi Shetty

Cover Work

Kirtee Shingan

About the Author

Philip Herron is an avid software engineer who focuses his passion towards 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 optimization of dynamic languages to develop a proof of concept GCC frontend to compile Python. This project sparked his deep interest of how Python works.

After completing a consecutive year on the same project in 2011, Philip decided to apply for 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 to add more high-level interfaces such as embedding Twisted web servers for REST calls to a system-level piece of software without writing any C code.

Currently Philip is employed by NYSE Euronext in Belfast Northern Ireland, working on multiprocessing systems. But he spends his evenings hacking on GCCPy, Cython, and GCC. In the past, he has worked with WANdisco as an Apache Hadoop developer and as an intern with SAP Research on cloud computing.

To achieve this book, I would like to thank many people. Firstly, my girlfriend Kirsty Johnston for putting up with my late nights and giving me the confidence I needed; you're the best! My mum and dad, Trevor and Ann Herron, who have always supported me my whole life; thanks for helping me so much.

I feel that Ian Lance Taylor from my GCC Google Summer of Code experience deserves a special mention; if it wasn't for you, I wouldn't be writing anything like this right now; you have shown me how to write software. Robert Bradshaw for mentoring my Cython GCC-PXD project even though I had a lot going on at the time; you helped me get it done and passed; you taught me how to manage time.

Special thanks Nicholas Marriott for helping me with the Tmux code base! I would also like to thank Gordon Hamilton, Trevor Lorimer, Trevor Thompson, and Dr Colin Turner for the support you've all given me.

About the Reviewers

Namit Kewat is a financial analyst and XBRL expert. At his job, he has worked on almost all the major SEC filers' XBRL creation (for example, BAC, GS, FB, and WSH). He is using Python extensively for extracting and generating reports from financial information present in XBRL financial reports. He has made a few quality checking apps in Python that are extensively used by his company for quality checks, which reduces the quality-check time from hours to seconds.

Goran Milovanovic is a Python programmer from the Blender Game Engine community. His interests include real-time simulation, nanotechnology, and education. If he is well known, it would be for his video tutorials, which can be found by Googling for "Goran's Python tutorial series".

I would like to thank my mother and father for their continuing support and encouragement.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
<hr/>	
Chapter 1: Cython Won't Bite	7
<hr/>	
What this book is	7
What this book isn't	8
Installing Cython	8
The emacs mode	8
Getting the code examples	9
Hello World	9
Module on your own	10
Calling into your C code	10
Type conversion	12
Summary	13
Chapter 2: Understanding Cython	15
<hr/>	
Cython cdef	15
Linking models	16
The public keyword	17
Logging into Python	18
Python ConfigParser	20
Cython cdef syntax and usage reference	21
Structs	22
Enums	24
Typedef and function pointers	25
Scalable asynchronous servers	26
C sockets with libevent	26
What is libevent?	26
Messaging engine	28
Cython callbacks	28
Cython PXD	28

Python messaging engine	29
Integration with build systems	31
Python distutils	31
GNU/Autotools	32
Summary	33
Chapter 3: Extending Applications	35
Cython pure Python code	35
Python bindings	36
Python garbage collector	37
Extending Tmux	38
Tmux build system	40
Embedding Python	42
Cythonizing struct cmd_entry	43
Implementing a Tmux command	46
Hooking everything together	47
Compiling pure Python code	49
Summary	50
Chapter 4: Debugging Cython	51
Using GDB on your code	51
Running cygdb	52
General Cython caveats	54
Type checking	55
No * operator	55
Python exceptions in C	56
For loops on C types	57
Bool type	58
No C const	59
Multiple Cython files	59
Initializing struct	59
Calling into pure Python modules	60
Keeping call stacks small and pure	60
Summary	60
Chapter 5: Advanced Cython	61
C++ constructs	61
Namespaces	61
Classes	62
C++ new keyword and allocation	63
Exceptions	64
Bool type	66
Overloading	66

Templates	67
Static class member attribute	68
Caveat on C++ usage	68
Calling in C and C++ functions	68
Namespaces	69
Python distutils	69
Python threading and GIL	69
Atomic instructions	70
Read/write lock	70
Cython keywords	71
Messaging server revisited	71
More inspiration	74
Messaging server working with SQL	75
Python IRC notifier	75
Unit testing the native code	75
Preventing subclassing	77
Cython typing via annotations	77
Parsing large amounts of data	78
Summary	81
Chapter 6: Further Reading	83
Keyword cpdef	83
OpenMP support	84
Object initialization	84
Compile time	84
Python 3	85
Using PyPy	85
AutoPXD	86
Pyrex versus Cython	86
SWIG versus Cython	86
Cython and NumPy	87
Numba versus Cython	88
Parakeet and Numba	89
GCCPy Python frontend to GCC	89
Links and further reading	90
Summary	90
Index	91

Preface

Cython is a tool that makes writing C extensions to Python as easy as writing Python itself. This is the slogan to which Cython conforms. For those who don't know what I am talking about, writing C extensions to Python from scratch is a fairly difficult process; unless you really understand the Python-C API fully with respect to GIL and garbage collection as well as managing your own reference counting, it's a very difficult process.

I tend to consider Cython to be along these lines: what Jython is to Java and Python, Cython is to C/C++ and Python. It allows us to extend and develop bindings to applications in a really intuitive manner so that we are able to reuse code from levels of the software stack. The Cython compiler compiles the Cython language or even pure Python to a native C Python module, which can be loaded like any Python module via the normal import. It not only generates all the wrapper and boilerplate code, but also commands the Python garbage collector to add all the necessary reference counting code.

What's interesting with the Cython language is that it has native support for understanding C types and is able to juggle them from both languages. It's simply an extension of Python that has additional keywords and some more constructs and which allows you to call into C or Python.

What this book covers

Chapter 1, Cython Won't Bite, will give you an introduction to what Cython is and how it works. It covers setting up your environment and running the "Hello World" application.

Chapter 2, Understanding Cython, will start to get serious with Cython and will discuss how to describe C declarations with respect to Cython along with calling conventions and type conversion.

Chapter 3, Extending Applications, will walk you through comparing the execution of pure Python code with the Cython version of the same code. We also look at extending Tmux, a pure C project, with Cython.

Chapter 4, Debugging Cython, will cover how to use GDB to debug your code and the relative GDB commands. There is also an extensive section on caveats and things to be aware of as well as conventions.

Chapter 5, Advanced Cython, will cover the usage of C++ with Cython, which is just as easy as using C with Cython. We will also work through all the syntax necessary to wrap C++. We will then look into the caveats and more on optimizations, comparing a Python XML parser with a Cython XML parser on large XML files.

Chapter 6, Further Reading, wraps up the book with a final look at some caveats and conventions. Then, we compare Cython against other similar tools like Numba and SWIG, and we will discuss how its used in NumPy and how we can use PyPy and Python 3.

What you need for this book

For this book, I used my MacBook and an Ubuntu virtual machine (GDB is too old on Mac OS X for debugging). You will require the following on Mac OS X:

- Xcode
- Cython
- GCC/Clang
- Make
- Python
- Python config
- Python distutils

On Ubuntu, you can install most components via the following:

```
$ sudo apt-get install build-essential gdb cython
```

Of course, I will go over this in the introduction, but as long as you have a C compiler and Python and have Python headers installed, you will have everything you need for Cython.

Who this book is for

This book is intended for C developers who like using Python and Python users wanting to implement native C/C++ extensions to Python. As a reader, you can expect to be shown how you can develop applications with Cython, with an emphasis on extending existing systems with help on how you can approach it.

Extending legacy systems can be difficult, but the rewards can be great. Consider very low-level system daemons that we could abstract and extend them and interact with the data from Python in a nice high-level way while leaving all performance-sensitive code alone! This model of development can prove to be efficient and of great return to development time; this can be particularly expensive when it comes to C applications.

It also allows for much more rapid development of the state or logic in a system. There is no need to worry about long data conversion algorithms in C for doing small things and then needing to change it all again.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "But note that you cannot use `del` on this instance else you will get an error."

A block of code is set as follows:

```
#ifndef __MY_HEADER_H__
#define __MY_HEADER_H__

namespace mynamespace {
    void myFunc (void);

    class myClass {
    public:
        int x;
        void printMe (void);
    };
}

#endif //__MY_HEADER_H__
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#ifndef __MY_HEADER_H__
#define __MY_HEADER_H__


namespace mynamespace {
    void myFunc (void);


    class myClass {
    public:
        int x;
        void printMe (void);
    };
}

#endif //__MY_HEADER_H__
```

Any command-line input or output is written as follows:

```
philips-macbook:primes redbrain$ cython pyprimes.py -embed
philips-macbook:primes redbrain$ gcc -g -O2 pyprimes.c -o pyprimes
`python-config --includes -libs`
```

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Cython Won't Bite

Cython makes writing C extensions for Python as easy as Python itself. Its main use within the community is the Mathematics software package, SAGE, which is used to perform fast and scalable calculations. Most notably, it provides a safe and maintainable way of building native modules for Python via autogeneration of the required code.

Personally, I have used Cython to take control of legacy applications where the system has been implemented in C/C++ and in which adding functionality can become painful; we can then use it to generate bindings so that the native application and the Python one can work together! With this, you are able to perform high-level logic within Python but leverage the power of your native system.

What this book is

Python has become a great exception in software engineering in the last few years; it can be used in any way you can think of to create or extend software systems with low cost in regards to development time. We can also use it to extend software ranging from system-level distributed systems to high-level web applications.

This book will demonstrate how to gain more from Python. In case you're not aware, Python can be extended via native C/C++ code using extension modules over **PyObject** or by using C types. Doing this manually is generally not a good idea, as you really need to know how Python works internally. For example, you need to know about garbage collection so your Python objects don't get collected. But this is where Cython comes in; it will generate all of the C Python API wrapper code necessary and correctly.

What this book isn't

It's good to be clear that in this book, I will assume you have experience and knowledge of C and Python, but more importantly, you should be comfortable with the C compilation and linking process to create shared libraries and executables. This is important to get the most out of Cython because the examples seen on the Internet generally deal with very small single Cython file projects, and those aren't that helpful for most of us. I hope that after reading this book you will be comfortable with Cython. The online documentation will provide all the references you will need.

Installing Cython

Now let's get Cython installed. Think of Cython as a tool like Bison, flex, or GCC; it takes an input source and generates another that you compile and link:

- **Fedora** – Fedora comes with the yum package manager. So, you can simply `run yum install Cython`.
- **Ubuntu/Debian** – As with Fedora, Ubuntu has a package available via aptitude: `apt-get install Cython`.
- **Mac** – Install Xcode and the command-line tools. Then, run the following:

```
$ curl -O http://www.cython.org/release/Cython-0.18.tar.gz

$ tar zxvf Cython-0.18.tar.gz

$ cd Cython-0.18
$ sudo python setup.py install
```
- **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>.

The emacs mode

There is an emacs mode for Cython available, as the Python emacs mode doesn't work correctly. So, you can add the `Tools/cython-mode.el` mode to your `~/.emacs.d` directory and then add `require` to your `~/.emacs` file.

```
(add-to-list 'load-path "~/.emacs.d/")
(require 'cython-mode)
```

Getting the code examples

Throughout this book, I intend to show real examples that are easy to digest to help you get a feel of the different things you can achieve with Cython. To access and download the code used in these examples, visit GitHub at the following link:

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

Hello World

Hopefully by now you've got Cython down and compiled and installed it. Let's check this by running the following command:

```
$ cython --version
```

Let's do a sanity test and run the typical "Hello World" program:

```
redbrain@gamma:~/workspace/cython-book/chapter1/helloworld$ make
```

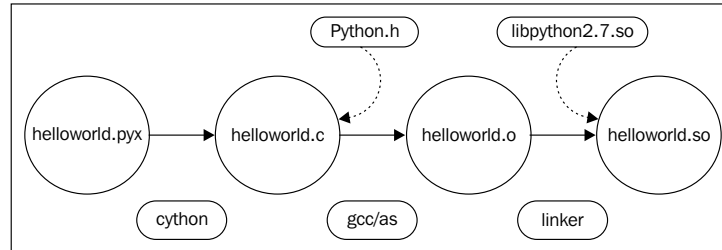
We have now created the Cython `helloworld.so` module! You can see it within Python (make sure you are in the same directory as the `helloworld.so` module):

```
redbrain@gamma:~/workspace/cython-book/chapter1/helloworld$ 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!
```

We import `helloworld` since this module is now a valid Python module that can be loaded. And, on importing, we declared the Cython code to simply print our message. Not very exciting, but that's how the "Hello World" module is.

Let's look at what we did; Cython files have the extensions `.pyx` and `.pxd`. For now, all we will care about are the `.pyx` files. Later in this book, I will introduce the use of `.pxd` and what you can use this for. For purposes of clarity, it's good to understand the basic pipeline of what's going on to generate this `helloworld.so` module for Python. Cython works in the same way as any other code generator.

The flow depicted in the following figure illustrates how Cython works:



I wrote a basic Makefile so you can simply run `make` to compile these examples. It uses the `setup.py` style to let Python handle compiling and setting up these modules. Here's the code to do this manually:

```
$ cython helloworld.pyx
$ gcc -g -O2 -fPIC `python-config --cflags` -c helloworld.c -o
helloworld.o
$ gcc -shared -o helloworld.so helloworld.o `python-config --libs`
```

I feel this is a very important skill to learn with C development because you will start thinking of your code in terms of how you can share it more easily.

Module on your own

Now that you've seen the "Hello World" module, let's see how you can write your own module to do something! Then, you can link it against some of your own code. Later, we'll introduce the idea of wrapping your code.

Calling into your C code

Cython is a superset of Python. Although the syntax and keywords will work in the same way, we should be careful when talking about Python and Cython for clarity. To see Cython in action, let's build a hello-world-style module but perform something basic, just to be sure we are on the same page.

Open a file called `mycode.c` and insert the following code in to it:

```
#include <stdio.h>

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

This is the C code we will call—just a simple function to add two integers you've probably seen before. Now let's get Python to call it. Open a file called `mycode.h`, wherein we will declare our prototypes for Cython as follows:

```
#ifndef __MYCODE_H__
#define __MYCODE_H__
extern int myfunc (int, int);
#endif //__MYCODE_H__
```

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.

Open a file called `mycodecpy.pyx` and insert the following code in to it:

```
cdef extern from "mycode.h":
    cdef int myfunc (int, int)

def callCfunc ():
    print myfunc (1,2)
```

Within this Cython code, we initially have to declare what C code we care about. `cdef` is a keyword signifying that this is from the C code that will be linked in. Now that we have declared the header with the prototype to squash any undeclared function warnings from our compiler, we can make a wrapper function. At this point, we will specify how Python will call this native code, since calling directly into C code is dangerous. Therefore, Cython handles all type-conversion problems for us. A basic wrapper function, `callCfunc`, is all we need—it calls the `myfunc` function and passes the integers 1 and 2; then it simply prints the result.

To compile this, use the following:

```
$ cython mycodecpy.pyx
$ gcc -g -O2 -fPIC -c mycode.c -o mycode.o
$ gcc -g -O2 -fPIC -c mycodecpy.c -o mycodecpy.o `python-config --cflags`
$ gcc -shared -o mycodecpy.so mycode.o mycodecpy.o `python-config --libs`
```

We have to remember to link in the code that has the C function; in this case, `mycode.c`. If you're not familiar with what I mean here, you may need to revisit some C tutorials on compilation, as every C file is compiled to an object file and then we link all object files into a binary. Therefore, you need to be sure you link in all necessary object files.

```
redbrain@gamma:~/workspace/cython-book/chapter1/ownmodule$ 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.
>>> from mycodepy import callCfunc
>>> callCfunc ()
look we are within your c code!!
3
```

So, we have now compiled and called our native code from Python code. I hope this gives you an idea of what you can do with it. With this, you can have a native module linking against some native libraries, making bindings to such libraries very simple.

Type conversion

You might have noticed that we called a Cython function directly with no arguments, which in turn called our C (the `cdef` prototype) function with two integer arguments. What if we wanted the Cython code to handle arguments? We could execute the following:

```
def callCfunc2 (int x, int y):
    print myfunc (x, y)
```

We have now added the `int` arguments to the Python wrapper function we defined. This will require Python code to be type-safe and to convert PyObjects to C types for us automatically. When you create an integer Python object, the type is not integer, it's PyObject. If you want to use this in C, you need to get the data via the Python C API, but Cython will do this for us automatically. For example, if you pass illegal arguments, you will get the following:

```
>>> import mycodepy
>>> mycodepy.callCfunc2 (1, 'string')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mycodepy.pyx", line 7, in mycodepy.callCfunc2 (mycodepy.c:733)
    def callCfunc2 (int x, int y):
TypeError: an integer is required
>>>
```

Even if you simply add more type safety to your Python code via the use of C types from Cython, you will find that you gain a bit of speed and some nice code. This is because the Cython compiler can optimize much more aggressively to avoid using Python calls.

Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Summary

Right, so I know this isn't the most exciting thing you've ever done, but if you could just take a step back and consider this: pure Python code calling into C code directly! Think what you could do with it—it's pretty exciting! In the next chapter, I will show you more on `cdef` and the keywords around it and how to share your Cython functions so that they are callable from C code. With this, anything can call anything! Not only that, we can look at how we compile our normal Python code and tell it to use some C types to try and get some more efficiency by looking into more syntax and more ways to use Cython.

2

Understanding Cython

If you were to create an API for Python, you should write it using Cython to create a more type-safe Python API. Or, you could take the C types from Cython to implement the same algorithms in your Python code, and they will be faster because you're specifying the types and you avoid a lot of the type conversion required.

Consider you are implementing a fresh project in C. There are a few issues we always come across in starting fresh; for example, choosing the logging or configuration system we will use or implement.

With Cython, we can reuse the Python logging system as well as the `ConfigParser` standard libraries from Python in our C code to get a head start. If this doesn't prove to be the correct solution, we can chop and change easily. We can even extend and get Python to handle all `getopt` usage. Since the Python API is very powerful, we might as well make Python do as much as it can to get us off the ground. Another question is do we want Python be our "driver" (main entry function) or do we want to handle this from our C code?

Cython cdef

In the next two examples, I will demonstrate how we can reuse the Python `logging` and Python `ConfigParser` modules directly from C code. But there are a few formalities to get over first, namely the Python initialization API and the link load model for fully embedded Python applications for using the shared library method used in *Chapter 1, Cython Won't Bite*.

It's very simple to embed Python within a C/C++ application; you will require the following boilerplate:

```
#include <Python.h>

int main (int argc, char ** argv)
```

```
{  
    Py_SetProgramName (argv [0]);  
    Py_Initialize ();  
    /* Do all your stuff in side here...*/  
    Py_Finalize ();  
    return 0;  
}
```

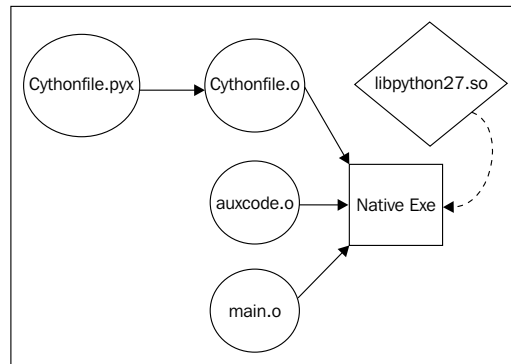


Make sure you always put the `Python.h` header at the very beginning of each C file, because Python contains a lot of headers defined for system headers to turn things on and off to make things behave correctly on your system.

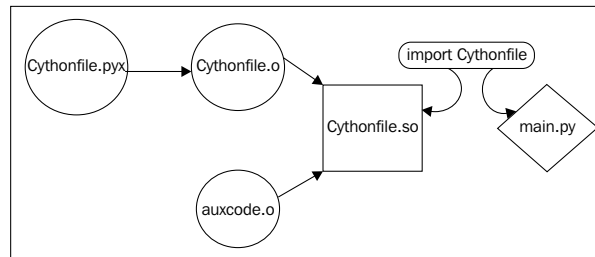
Later, I will introduce some important concepts about the **GIL** that you should know and the relevant Python API code you will need to use from time to time. But for now, these few calls will be enough for you to get off the ground.

Linking models

Linking models are extremely important when considering how we can extend or embed things in native applications. There are two main linking models for Cython: fully embedded Python and code, which looks like the following figure:



This demonstrates a fully embedded Python application where the Python runtime is linked into the final binary. This means we already have the Python runtime, whereas before we had to run the Python interpreter to call into our Cython module. There is also a Python shared object module as shown in the following figure (which is what we did in the previous chapter):



We have now fully modularized Python. This would be a more Pythonic approach to Cython, and if your code base is mostly Python, this is the approach you should take if you simply want to have a native module to call into some native code, as this lends your code to be more dynamic and reusable.

The public keyword

Moving on from linking models, we should next look at the `public` keyword, which allows Cython to generate a C/C++ header file that we can include with the prototypes to call directly into Python code from C.

The main caveat if you're going to call Python `public` declarations directly from C is if your link model is fully embedded and linked against `libpython.so`; you need to use the boilerplate code as shown in the previous section. And before calling anything with the function, you need to initialize the Python module example if you have a `cythonfile.pyx` file and compile it with `public` declarations such as the following:

```
cdef public void cythonFunction ():
    print "inside cython function!!!"
```

You will not only get a `cythonfile.c` file but also `cythonfile.h`; this declares a function called `extern void initcythonfile (void)`. So, before calling anything to do with the Cython code, use the following:

```
/* Boiler plate init Python */
Py_SetProgramName (argv [0]);
Py_Initialize ();
/* Init our config module into Python memory */
```

```
    initpublicTest ();  
    cythonFunction ();  
  
    /* cleanup python before exit ... */  
    Py_Finalize ();
```

Calling `initcythonfile` can be considered as the following in Python:

```
import cythonfile
```

Just like the previous examples, this only affects you if you're generating a fully embedded Python binary.

Logging into Python

A good example of Cython's abilities in my opinion is reusing the Python `logging` module directly from C. So, for example, we want a few macros we can rely on, such as `info (...)` that can handle `VA_ARGS` and feels as if we are calling a simple `printf` method.

I think that after this example, you should start to see how things might work when mixing C and Python now that the `cdef` and `public` keywords start to bring things to life:

```
import logging  
  
cdef public void initLogging (char * logfile):  
    logging.basicConfig (filename = logfile,  
                        level = logging.DEBUG,  
                        format = '%(levelname)s %(asctime)s:  
                                %(message)s',  
                        datefmt = '%m/%d/%Y %I:%M:%S')  
cdef public void pyinfo (char * message):  
    logging.info (message)  
  
cdef public void pydebug (char * message):  
    logging.debug (message)  
  
cdef public void pyerror (char * message):  
    logging.error (message)
```

This could serve as a simple wrapper for calling directly into the Python logger, but we can make this even more awesome in our C code with C99 `__VA_ARGS__` and an attribute that is similar to GCC `printf`. This will make it look and work just like any function that is similar to `printf`. We can define some headers to wrap our calls to this in C as follows:

```

#ifndef __MAIN_H__
#define __MAIN_H__

#include <Python.h>

#include <stdio.h>
#include <stdarg.h>

#define printflike \
    __attribute__ ((format (printf, 3, 4)))

extern void printflike cinfo (const char *, unsigned, const char *,
...);
extern void printflike cdebug (const char *, unsigned, const char *,
...);
extern void printflike cerror (const char *, unsigned, const char *,
...);

#define info(...) \
    cinfo (__FILE__, __LINE__, __VA_ARGS__)

#define error(...) \
    cerror (__FILE__, __LINE__, __VA_ARGS__)

#define debug(...) \
    cdebug (__FILE__, __LINE__, __VA_ARGS__)

#include "logger.h" // remember to import our cython public's

#endif // __MAIN_H__

```

Now we have these macros calling `cinfo` and the rest, and we can see the file and line number where we call these logging functions:

```

void cdebug (const char * file, unsigned line,
             const char * fmt, ...)
{
    char buffer [256];
    va_list args;

```

```
    va_start (args, fmt);
    vsprintf (buffer, fmt, args);
    va_end (args);

    char buf [512];
    snprintf (buf, sizeof (buf), "%s-%i -> %s",
              file, line, buffer);
    pydebug (buf);
}
```

On calling `debug ("debug message")`, we see the following output:

```
Philips-MacBook:cpy-logging redbrain$ ./example log
Philips-MacBook:cpy-logging redbrain$ cat log
INFO 05/06/2013 12:28:24: main.c-62 -> info message
DEBUG 05/06/2013 12:28:24: main.c-63 -> debug message
ERROR 05/06/2013 12:28:24: main.c-64 -> error message
```

Also, you should note that we import and do everything we would do in Python as we would in here, so don't be afraid to make lists or classes and use these to help out. Remember if you had a Cython module with `public` declarations calling into the logging module, this integrates your applications as if it were one.

More importantly, you only need all of this boilerplate when you fully embed Python, not when you compile your module to a shared library.

Python ConfigParser

Another useful case is to make Python's `ConfigParser` accessible in some way from C; ideally, all we really want is to have a function to which we pass the path to a config file to receive a `STATUS OK/FAIL` message and a filled buffer of the configuration that we need:

```
from ConfigParser import SafeConfigParser, NoSectionError
cdef extern from "main.h":
    struct config:
        char * path
        int number
    cdef config myconfig
```

Here, we've Cythoned our struct and declared an instance on the stack for easier management:

```
cdef public config * parseConfig (char * cfg):
    # initialize the global stack variable for our config...
```

```

myconfig.path = NULL
myconfig.number = 0
# buffers for assigning python types into C types
cdef char * path = NULL
cdef number = 0
parser = SafeConfigParser ()
try:
    parser.readfp (open (cfg))
    pynumber = int (parser.get ("example", "number"))
    pypath = parser.get ("example", "path")
except NoSectionError:
    print "No section named example"
    return NULL
except IOError:
    print "no such file ", cfg
    return NULL
finally:
    myconfig.number = pynumber
    myconfig.path = pypath
return &myconfig

```

This is a fairly trivial piece of Cython code that will return `NULL` on error as well as the pointer to the struct containing the configuration:

```

Philips-MacBook:cpy-configparser redbrain$ ./example sample.cfg
cfg->path = some/path/to/something
cfg-number = 15

```

As you can see, we easily parsed a config file without using any C code. I always found figuring out how I was going to parse config files in C to be a nightmare. I usually ended up writing my own mini domain-specific language using Flex and Bison as a parser as well as my own middle-end, which is just too involved.

Cython cdef syntax and usage reference

So far, we have explored how to set up Cython and how to run "Hello World" modules. Not only that, we have also seen how we can call our own C code from Python. Let's take a look at how we can interface Python into different C declarations such as `structs`, `enums`, and `typedefs`. We will use this to build up a cool project at the end of the chapter.

Although not that interesting or fun, this small section should serve as a reference for you later on when you're building your next awesome project.

Structs

Let's begin by creating a C struct. Open `mycode.h`:

```
#ifndef __MYCODE_H__
#define __MYCODE_H__

struct mystruct {
    char * string;
    int integer;
    char ** string_array;
};

extern void printStruct (struct mystruct *);

#endif //__MYCODE_H__
```

Now we can use Cython to interface and initialize structs and even allocate/free memory. There are a few pointers to make a note of when doing this, so let's create the code. First we need to create the Cython declaration:

```
cdef extern from "mycode.h":
    struct mystruct:
        char * string
        int integer
        char ** string_array
    void printStruct (mystruct *)

def testStruct ():
    cdef mystruct s
    cdef char *array [2]
    s.string = "Hello World"
    s.integer = 2
    array [0] = "foo"
    array [1] = "bar"
    s.string_array = array
    printStruct (&s)
```

Let's look at this line by line. First off, we see the `cdef` keyword; this tells Cython that this is an external C declaration and that the original C declarations can be included from `mycode.h`; the generated code from Cython can include this to squash all warnings about undeclared symbols. Anything that is within this `cdef` suite, Cython will treat as a `cdef`. The struct looks very similar to normal C structs—just be careful with your indentation. Also be sure, even in the `cdef` functions, that if you want explicit C types, you need to declare this with the `cdef` type identifier to make sure they will be of the correct type and not just PyObjects.

The final caveat with structs is when defining a `cdef` declaration for a function. If a parameter is a struct, you never declare it as:

```
void myfunc (struct mystruct * x)
```



Instead, we simply use the following:

```
void myfunc (mystruct * x)
```

Cython will figure it out.

There are a few subtleties with the `testStruct` function. We declare our struct and array on the stack with `cdef` as well, as this allows us to declare variables. In Cython, we have the reference operator `&`; this works just as in C, so we have the struct on the stack and we can pass a pointer via the reference operator just like in C. But we *don't* have a `->` operator in Cython, so when trying to access the struct (even if it is on a pointer), we simply use the `.` operator. Cython understands this at compile time. We also have an extension in Cython to specify fixed length arrays as shown and assignment should look very familiar. A simple `makefile` for this system would be as follows:

```
all:
    cython -2 -o mycodepy.c mycodepy.pyx
    gcc -g -O2 -fpic -c mycodepy.c -o mycodepy.o `python-config
--cflags`
    gcc -g -O2 -fpic -c mycode.c -o mycode.o
    gcc -g -O2 -shared -o mycodepy.so mycode.o mycodepy.o

clean:
    rm -f *.o *.so *~ mycodepy.c
```

And a simple `printStruct` function would be as follows:

```
#include <stdio.h>
#include "mycode.h"

void printStruct (struct mystruct * s)
{
    printf (".string = %s\n", s->string);
    printf (".integer = %i\n", s->integer);
    printf (".string_array = \n");

    int i;
    for (i = 0; i < s->integer; ++i)
        printf ("\t[%i] = %s\n", i, s->string_array [i]);
}
```

A simple run of this in the downloaded code is as follows:

```
redbrain@blue-sun:~/workspace/cython-book/chapter2/c-decl-reference$ make
cython -2 -o mycodepy.c mycodepy.pyx
gcc -g -O2 -fpic -c mycodepy.c -o mycodepy.o `python-config --cflags`
gcc -g -O2 -fpic -c mycode.c -o mycode.o
gcc -g -O2 -shared -o mycodepy.so mycode.o mycodepy.o
```

```
redbrain@blue-sun:~/workspace/cython-book/chapter2/c-decl-reference$
python
Python 2.7.3 (default, Sep 26 2012, 21:51:14)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from mycodepy import testStruct
>>> testStruct ()
.string = Hello World
.integer = 2
.string_array =
  [0] = foo
  [1] = bar
```

This simply demonstrates that Cython can work properly with C structs – it initialized the C struct and assigned it data correctly, as you would expect if it was from C.

Enums

Interfacing with C enums is simple. If you have the following enum in C:

```
enum cardsuit {
    CLUBS,
    DIAMONDS,
    HEARTS,
    SPADES
};
```

This can be expressed as the following Cython declaration:

```
cdef enum cardsuit:
    CLUBS, DIAMONDS, HEARTS, SPADES
```

Then, use the following as the `cdef` declaration within our code:

```
cdef cardsuite card = CLUBS
```

Typedef and function pointers

Typedefs are just how you would expect them to be. It's simpler to understand with examples; consider the following C code:

```
struct foobar {
    int x;
    char * y;
};
typedef struct foobar foobar_t;
```

In Cython, this can be described by the following:

```
cdef struct foobar:
    int x
    char * y
ctypedef foobar foobar_t
# You can also typedef pointers too

ctypedef int * int_ptr
```

We can also typedef function pointers as follows:

```
typedef void (*cfptr) (int)
```

In Cython, this will be as follows:

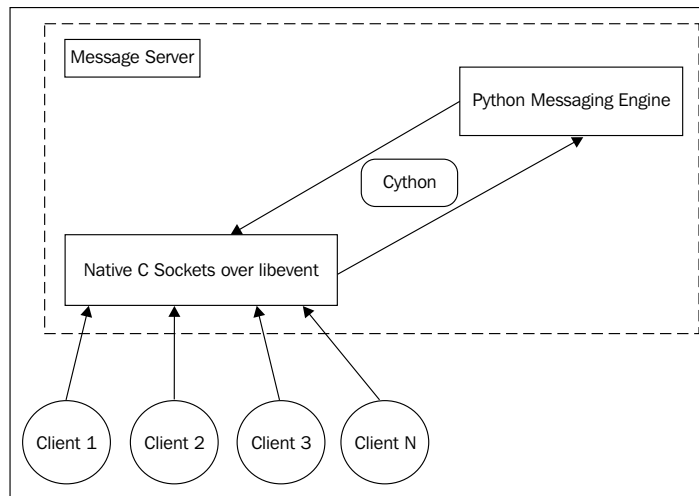
```
ctypedef void (*cfptr)(int)

# then we use the function pointer:
cdef cfptr myfunctionptr = &myfunc
```

Overall, this should be the reference you use whenever you are using Cython to understand how your C types map to Cython and to see how you can use them.

Scalable asynchronous servers

Using all the concepts learned in this chapter, I want to show you how we can use Cython to build something awesome – a complete messaging server that uses C to do all the low-level I/O and `libevent` to keep everything asynchronous. This means we will be using callbacks to handle the events that we will manage in the Python messaging engine. We can then define a simple protocol for a messaging system and roster. This design can be easily extended to a lot of things. To see if we are on the same page, refer to the following figure:



C sockets with libevent

For those of you who are unfamiliar with `libevent`, I will now give a brief overview and show the main parts of the code you should look at in `chapter2/async-server/server1/server.c`.

What is libevent?

`libevent` allows us to create a socket in C, which we can use to pass the file descriptor to `libevent` and give it several events to care about; for example, if a client is connecting to this socket, we can tell `libevent` to listen for it and it call our callback. Other events such as errors (clients going offline) or reads (clients pushing up data) can also be handled in the same manner. We use `libevent` because it's much more scalable and well defined, and it is a far better choice than writing our own polling event loop.

Once we create a socket, we must make it non-blocking for `libevent`. This useful snippet of C code may or may not be familiar to you, but it's a useful one to have in your tool-belt:

```
int setnonblock (int fd)
{
    int flags;
    flags = fcntl (fd, F_GETFL);
    if (flags < 0)
        return flags;
    flags |= O_NONBLOCK;
    if (fcntl (fd, F_SETFL, flags) < 0)
        return -1;
    return 0;
}
```

Once you create a socket, you pass the resulting file descriptor to this function and then create an on-connect event for `libevent`:

```
struct event ev_accept;
event_assign (&ev_accept, evbase,
             sockfd,
             EV_READ|EV_PERSIST,
             &callback_client_connect,
             NULL);
event_add (&ev_accept, NULL);
```

Now we have an event that will call the `callback_client_connect` function. Test this server with the following:

```
redbrain@blue-sun:~/workspace/cython-book/chapter2/async-server/server1$
make
gcc -g -O2 -Wall -c server.c -o server.o
gcc -g -O2 -o server server.o -levent
redbrain@blue-sun:~/workspace/cython-book/chapter2/async-server/server1$
./server
```

In another shell or multiple shells, run `telnet` to act as a simple client for now:

```
$ telnet localhost 9080
```

You can now type away and see all your data and events. At the moment, this is just a dumb event-driven messaging system, but imagine how you would begin adding a messaging engine to pass messages between clients and set how you would up a protocol in C. It would take some time to map out and, in general, it would be an unpleasant experience. We can use Cython to take control of the server and create our logic in Python using callbacks.

Messaging engine

With these callbacks, we can start making use of Python very easily to make this project awesome.

Cython callbacks

If you look at `cython-book/chapter2/async-server/server2`, you can see the callbacks in action:

```
./messagingServer -c config/server.cfg -l server.log
```


You can also spawn multiple telnet sessions again to see some things being printed out. There is a lot going on here, so I will break it down first. If you look inside this directory, you will see `pyserver.pyx` and `pyserver.pxd`. Here, we will introduce the pseudo Cython header files: (`*.pxd`).

Cython PXD

The use of PXD files is very similar to that of header files in C/C++. We can simply use our `cdef` declarations like `extern` functions or struct definitions and then use the following within a `*.pyx` file:

```
cimport pyserver
```

Now you can just code your method prototypes like you would in C and the `cimport` of the PXD file will get all the definitions.

 **Caveat**
Cython's input filenames *cannot* handle dashes, `-`, in their filenames. It's best to try and use camelcase, since you can't use `cimport my-import` in Python.

Now that you have seen how `*.pxd` files work, we will remove the `main` method from `server.c` so we can use Python to control the whole system. If you look at `pyserver.pyx`, you will see the `pyinit_server` function; it takes a port number. We can then from Python pass the configuration of the server from pure Python with `import pyserver` when we build the shared library. We also call `server.c` to set callbacks, which are the `cdef` Cython functions, and we pass their addresses to the server:

```
static callback conncb, discb, readcb;  
  
void setConnect_PyCallback (callback c)
```

```

{
    conncb = c;
}
void setDisconnect_PyCallback (callback c)
{
    discb = c;
}
void setRead_PyCallback (callback c)
{
    readcb = c;
}

```

Now, in each of the events that exist, we can call these callbacks simply with `readcb (NULL, NULL)` and we will be in Python land. You can look at the Cython functions in depth in the `pyserver.pyx` file; however, know that they just print out some data:

```

cdef void pyconnect_callback (client *c, char * args):
    print c.cid, "is online..."

cdef void pydisconnect_callback (client *c, char * args):
    print c.cid, "went offline..."

cdef void pyread_callback (client *c, char * args):
    print c.cid, "said: ", args

```

These are your basic callbacks into Cython code from the native event-driven system. You can see the basic `main` method from the `messageServer.py` file. It is executable and initializes everything required for our purposes. I know this may seem a fairly niche example, but I truly believe it demonstrates how cool C/Python can be. It simply imports `pyserver` and calls `pyinit_server` with a port. With this, you can use Python to control the configuration of system-level C components very easily, which can be fiddly to do well in pure C. We let Python do it.

Python messaging engine

Now that you've seen how we can have callbacks from this system into Cython, we can start to add some logic to the system so that if you spawn multiple localhost connections, they will run concurrently. It would be good to have some `Roster` logic, say to just make the client address its identifier, such that there can be only one client per address. We could implement this via a simple dictionary where key is address and value is `true` or `false` for online or offline. We can query if it is online; return a yes if it is or no to kill the connection. Currently, `messagingEngine.py` implements a basic `roster` class to perform this function.

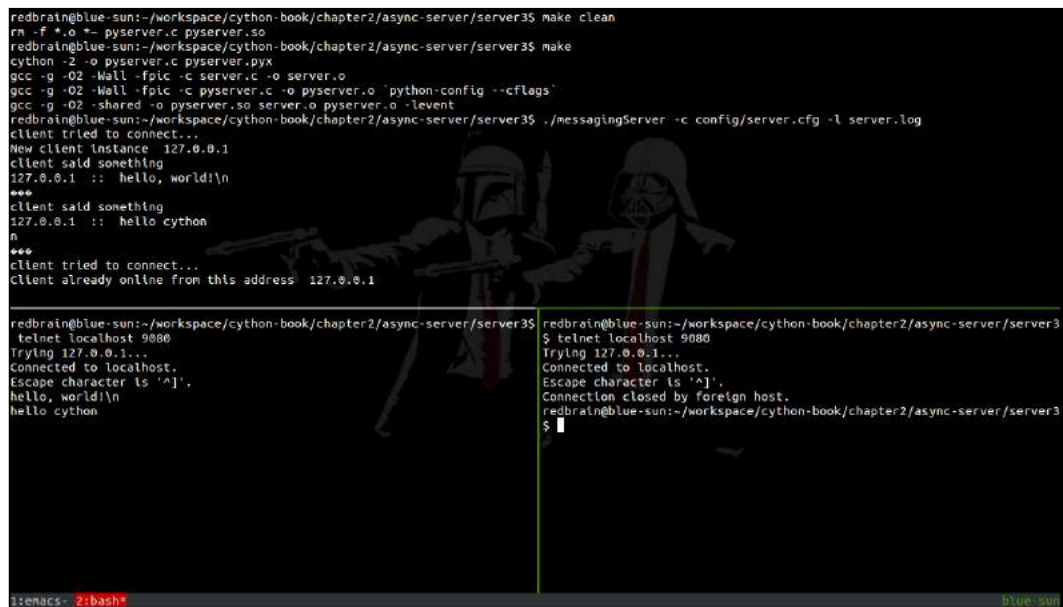
This `roster` class will initialize a dictionary of client objects against their name, and `handleEvent` will, if it's a `rosterEvent`, handle clients going online and offline via the Cython callbacks. The other case is if the client is already online. We return `true` if we want to tell the server to disconnect that client by closing the socket connection, else we return `false`.

A simple way to initialize the `roster` class is through `pyserver.pyx`:

```
from messagingEngine import Roster
roster = None

def pyinit_server (port):
    global roster
    roster = Roster ()
    ...
```

Now, in each of the callbacks, we can simply call `roster.handleEvent (...)`. On running this, we can see that the same address connections are now closed, as shown in the following screenshot (only one instance is allowed to personify clients logging in to a system):



I think this gives you an idea of how easy it could be to have Python handle message passing. You can easily extend your read callbacks to fully read the buffer and use Google protocol buffers (<https://developers.google.com/protocol-buffers/docs/pythontutorial>) to implement a full protocol for your system, but that's a whole project of its own.

Integration with build systems

This topic is basically dependent on the linking model you choose if you are to choose the shared-library approach. I would recommend using Python `distutils`. And if you are going for embedded Python, you should choose the autotools approach.

Python `distutils`

I just want to note how you can integrate Cython into your `setup.py` file; it's very simple:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    scripts = ['messagingServer.py'],
    packages = ['messagingEngine'],
    cmdclass = { 'build_ext' : build_ext },
    ext_modules = [ Extension ("pyserver", ["pyserver.pyx",
                                         "server.c" ]) ]
)
```

Just append your module sources and Cython picks up the `*.pyx` and `*.c` files. You can use `setup.py` as you normally would:

```
$ python setup.py build
$ python setup.py install
```

Note that to install correctly, you must package and modularize your project so that `messagingEngine` is now its own module:

```
$ mkdir messagingEngine
$ cd messagingEngine
$ mv ../messagingEngine.py .
$ touch __init__.py
$ $EDITOR __init__.py
__all__ = ['messagingEngine']
```

GNU/Autotools

The snippet you need to know for this would simply be as follows:

```
found_python=no
AC_ARG_ENABLE(
    python,
    AC_HELP_STRING(--enable-python, create python support),
    found_python=yes
)
AM_CONDITIONAL(IS_PYTHON, test "x$found_python" = xyes)

PYLIBS=""
PYINCS=""
if test "x$found_python" = xyes; then
    AC_CHECK_PROG(CYTHON_CHECK, cython, yes)
    if test x"$CYTHON_CHECK" != x"yes" ; then
        AC_MSG_ERROR([Please install cython])
    fi
    AC_CHECK_PROG(PYTHON_CONF_CHECK, python-config, yes)
    PYLIBS=`python-config --libs`
    PYINCS=`python-config --includes`
    if test "x$PYLIBS" == x; then
        AC_MSG_ERROR("python-dev not found")
    fi
fi
AC_SUBST(PYLIBS)
AC_SUBST(PYINCS)
```

This adds the `--enable-python` switch to your configure script. You now have the Cython command `found` and the `PYLIBS` and `PYINCS` variables for the compilation flags you need to compile. Now you need a snippet to understand how to compile `*.pyx` in your sources in automake:

```
bin_PROGRAMS = myprog
ACLOCAL_AMFLAGS = -I etc

CFLAGS += -I$(PYINCS)

LIBTOOL_DEPS = @LIBTOOL_DEPS@
libtool: $(LIBTOOL_DEPS)
    $(SHELL) ./config.status libtool
```

```
SUFFIXES = .pyx
.pyx.c:
    @echo " CPY    " $<
    @cython -2 -o $@ $<

myprog_SOURCES = \
    src/bla.pyx \
...
myprog_LDADD = \
    $(PYLIBS)
```

When you're comfortable with understanding where your code is and the linking models, you can choose the build systems. At that point, embedding Python becomes very easy – almost like second nature.

Summary

This whole chapter dealt with trying to make you more comfortable with Cython and aimed to show you that it is just like writing Python code. If you start using `public` and `cdef` regularly, you will see that you can mix C and Python code as if it was all the same language! Better yet, in each language, you get access to everything that language has. So, if you have Twisted installed in Python, you can access Twisted when you're in Python land; and if you're in C land, you can use `fcntl` or `ioctl`!

3

Extending Applications

As mentioned in previous chapters, I want to show how you can extend existing systems with Cython. So let's get right to doing that. We have several different approaches and techniques to use. Cython was originally designed to make computation faster with Python as part of the SAGE project. Therefore, you can actually plainly convert some of your Python code to use C types on big computations for an increase in speed. We can also, as we have seen, mix C and Python code to leverage extensive high-level Python APIs with the low-level system from C.

Cython pure Python code

Let's view a mathematical application that is actually taken from the Cython documentation. I wrote this equivalent in pure Python so we can compare the speed. If you open `chapter3/primes`, you will see two programs - the Cython `primes.pyx` example and my pure Python port. They both look almost the same:

```
def primes(kmax):
    n = 0
    k = 0
    i = 0
    if kmax > 1000:
        kmax = 1000
    p = [0] * kmax
    result = []
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
```

```
        p[k] = n
        k = k + 1
        result.append(n)
    n = n + 1
    return result
primes (10000)
```

This really is a direct Python port of that Cython code. Both call `primes (10000)`, but the evaluation time is very different for them in terms of performance:

```
Philips-MacBook:primes redbrain$ make
cython --embed primes.pyx
gcc -g -O2 -c primes.c -o primes.o `python-config --includes`
gcc -g -O2 -o primes primes.o `python-config -libs`
```

```
Philips-MacBook:primes redbrain$ make test
time python pyprimes.py
    0.18 real    0.17 user    0.01 sys
time ./primes
    0.04 real    0.03 user    0.01 sys
```

You can see that the pure Python version was almost five times slower in doing the exact same job. Moreover, nearly every line of code is the same. Cython can do this because we have explicitly expressed the C types, hence there is no type conversion or folding for Python to do. I just want to draw attention to the kind of speedups you can get with just simple code without calling into other native libraries.

Python bindings

Creating bindings to different native libraries is basically what we've been doing since the start, but it's good to iterate on how easy this can be for already-installed libraries.

When making bindings to projects, you must spend time analyzing the headers that you include in normal C applications. Once you do this, you're 80 percent of the way there. If you want, you can directly copy and implement the headers as a `.pxd` file and then just write Cython wrappers.

Although this approach is very formal and correct, it isn't really that helpful since you might as well just use C—and it doesn't even vaguely feel Pythonic. I would recommend writing wrapper functions that call the functions you care about yourself, making the library much easier to use.

Python garbage collector

Extending bindings to be aware of memory management can be pretty awesome – you can make the Python garbage collector handle all memory management for you! If you have a structure in C, you can use the hooks from Python to make it feel just like any Python type. Consider the following C structure:

```
typedef struct data {
    int val;
} data_t;
```

We can then write the Cython declaration of the C struct into `cdata.pxd` as follows:

```
cdef extern from "data.h":
    struct data:
        int val
    ctypedef data data_t
```

We have several Cython-specific hooks. The one we care about is `__cinit__`. It is called in a `cdef` class just before `__init__`. You cannot assume any self-initialization since `__init__` has not been called. But it allows us to do some memory allocation to allocate the C structure. Now consider the Python class implementation using the `__cinit__` hook for the initialization of your type:

```
cimport cdata
from libc.stdlib cimport malloc, free

cdef class Data:
    cdef cdata.data_t * _c_data # the pointer to the c struct

    # has the same function sig to __init__ both will be called

    def __cinit__(self, val):
        self._c_data = <data_t *> malloc (sizeof (data_t))
        if not self._c_data: raise MemoryError ()
        self._c_data.val = val
```

You might have noticed that we were able to call into `libc`, whose `.pxd` files are provided by Cython, and we used the `<>` syntax to cast the returned type `void *` from `malloc` to our correct C type.

Now you can simply use this as if it were a normal Python class and it will automatically be allocated memory at initialization pre `__init__`. What's more, we can now get Python to reuse the garbage collector on this type for us. So, consider the `__dealloc__` hook as follows:

```
def __dealloc__ (self):
    if self._c_data is not NULL:
        free (self._c_data)
        self._c_data = NULL
```

Now, when the normal Python reference counter garbage collector figures out we aren't using data anymore, it will automatically free the structure and the Python object. Also, if we want a nice print on the type, we can use the normal Python `__str__` hook:

```
cdef int getVal (self):
    if self._c_data: return self._c_data.val
    else: return -1

def __str__ (self):
    if self._c_data is not NULL:
        return "%s" % self.getVal ()
    else:
        return "Object not initialized!"
```

Now, if you're using Python, you don't need to care about any memory management hooks you need to call; just let the Python garbage collector handle everything!

Extending Tmux

Tmux is a terminal multiplexer inspired by GNU Screen (<http://tmux.sourceforge.net/>), but it supports much simpler and better configuration. More importantly, the implementation is much cleaner and easier to maintain, and it also uses `libevent` and very well-written C code.

I want to show you how you can extend Tmux with new built-in commands by writing Python code instead of C. Overall, there are several parts to this project, as follows:

- Hack the autotools build system to compile in Cython
- Create PXD declarations to the relevant declarations, such as `struct cmd_entry`
- Embed Python into Tmux
- Add the Python command to the global Tmux `cmd_table`

Let's take a quick look at the Tmux source, and in particular any of the `cmd-*.c` files that contain command declarations and implementations. Consider, for example, that `cmd-kill-window.c` is the command entry. This tells Tmux the name of the command, its alias, and how it may or may not accept arguments; finally, it accepts a function pointer to the actual command code:

```
const struct cmd_entry cmd_kill_window_entry = {
    "kill-window", "killw",
    "at:", 0, 0,
    "[-a] " CMD_TARGET_WINDOW_USAGE,
    0,
    NULL,
    NULL,
    cmd_kill_window_exec
};
```

So, if we are able to implement and initialize our own struct containing this information, we can run our `cdef` code. Next, we need to look at how Tmux picks up this command definition and how it gets executed.

If we look at `tmux.h`, we find the prototypes for everything we need to manipulate:

```
extern const struct cmd_entry *cmd_table[];
extern const struct cmd_entry cmd_attach_session_entry;
extern const struct cmd_entry cmd_bind_key_entry;
...
```

So, we need to add a prototype here for our `cmd_entry` definition. Next, we need to look at `cmd.c`; this is where the command table is initialized so that it can be looked up later on for executing commands:

```
const struct cmd_entry *cmd_table[] = {
    &cmd_attach_session_entry,
    &cmd_bind_key_entry,
    ...
};
```

Now that the command table is initialized, where does the code get executed? If we look at the `cmd_entry` definition in the `tmux.h` header, we can see the following:

```
/* Command definition. */
struct cmd_entry {
    const char *name;
    const char *alias;

    const char *args_template;
    int args_lower;
};
```

```
int      args_upper;

const char *usage;

#define CMD_STARTSERVER 0x1
#define CMD_CANTNEST 0x2
#define CMD_SENDENVIRON 0x4
#define CMD_READONLY 0x8
int      flags;

void      (*key_binding)(struct cmd *, int);
int      (*check)(struct args *);
enum cmd_retval (*execc)(struct cmd *, struct cmd_q *);
};
```

The `execc` hook is the function pointer we really care about, so if you `grep` the sources, you should find the following:

```
Philips-MacBook:tmux-project redbrain$ ack-5.12 execc
tmux-1.8/cmd-queue.c
229:             retval = cmdq->cmd->entry->execc(cmdq->cmd, cmdq);
```

You might notice that in the official Tmux Git, this hook is simply named `exec`. I renamed this to `execc` because `exec` is a reserved word in Python – we need to avoid things like that. To begin with, let's get some code compiled; firstly, we need to get the build system to play ball.

Tmux build system

Tmux uses autotools; we can reuse the snippets from *Chapter 2, Understanding Cython*, to add in Python support. We can add the `-enable-python` switch into `configure.ac` as follows:

```
# want python support for pytmux scripting
found_python=no
AC_ARG_ENABLE(
  python,
  AC_HELP_STRING(--enable-python, create python support),
  found_python=yes
)
AM_CONDITIONAL(IS_PYTHON, test "$found_python" = xyes)

PYLIBS=""
PYINCS=""
```

```

if test "x$found_python" = xyes; then
  AC_CHECK_PROG(CYTHON_CHECK, cython, yes)
  if test x"$CYTHON_CHECK" != x"yes" ; then
    AC_MSG_ERROR([Please install cython])
  fi
  AC_CHECK_PROG(PYTHON_CONF_CHECK, python-config, yes)
  PYLIBS=`python-config --libs`
  PYINCS=`python-config --includes`
  if test "x$PYLIBS" == x; then
    AC_MSG_ERROR("python-dev not found")
  fi
  AC_DEFINE(HAVE_PYTHON)
fi
AC_SUBST(PYLIBS)
AC_SUBST(PYINCS)

```

This gives us the `./configure --enable-python` option; next, we need to look at the `Makefile.am` file. Let's call our Cython file `cmdpython.pyx`; note that Cython doesn't like awkward characters like `-` in the filename, as explained in *Chapter 2, Understanding Cython*. If we are to make Python support a conditional option at build time, we should add the following to `Makefile.am`:

```

if IS_PYTHON
PYTHON_SOURCES = cmdpython.pyx
else
PYTHON_SOURCES =
endif

# List of sources.
dist_tmux_SOURCES = \
  $(PYTHON_SOURCES) \
  ...

```

We make sure it is needed and compiled first. Remember that if we create public declarations, Cython generates a header for us; we will simply add our public header to `tmux.h` to keep headers very simple. Then, to make sure Cython files get picked up by automake and compiled properly according to the correct dependency management at build time, we need to add in the following:

```

SUFFIXES = .pyx
.pyx.c:
  @echo "  CPY    " $<
  @cython -2 -o $@ $<

```

This adds in the suffix rule to make sure the *.pyx files are Cythoned and then told to compile the resulting .c file just as any normal C file. This snippet plays well if you happen to use `AM_SILENT_RULES([yes])` in your autotools project, which formats the echo message correctly. Lastly, we need to make sure we add the necessary `CFLAGS` and `LIBS` options to the compiler from `AC_SUBST` in the configure script.

```
CFLAGS += $(PYINCS)
tmux_LDADD = \
    $(PYLIBS)
```

Now you should have everything ready in the build system, but we have to regenerate the autotools stuff now because of the changes made. Simply run `./autogen.sh`.

Embedding Python

Now that we have files being compiled, we need to initialize Python and our module. Tmux is a forked server that clients connect to, so try not to think of it as a single-threaded system. It's a client *and* a server, so all commands are executed on the server. Now let's find where the event loop is started in the server and initialize and finalize the server here so that it's done correctly. Looking at `int server_start(int lockfd, char *lockfile)`, we can add in the following:

```
#ifdef HAVE_PYTHON
    Py_InitializeEx (0);
#endif
    server_loop();
#ifdef HAVE_PYTHON
    Py_Finalize ();
#endif
```

Python is now embedded into the Tmux server. Notice that instead of simply `Py_Initialize`, I used `Py_InitializeEx (0)`. This replicates the same behavior but doesn't start up normal Python signal handlers. Tmux has its own signal handlers, so I don't want to override them. It's probably a good idea when extending established applications such as this to use `Py_InitializeEx (0)` since they generally implement their own signal handling. Using this stops Python trying to handle signals which would conflict.

Cythonizing struct cmd_entry

Next, let's consider creating a `cythonfile.pxd` file for the necessary `cdef` declarations of Tmux that we need to be aware of. We need to look at the `struct cmd_entry` declaration and work backwards from this:

```

struct cmd_entry {
    const char *name;
    const char *alias;

    const char *args_template;
    int     args_lower;
    int     args_upper;

    const char *usage;
    int     flags;

    void      (*key_binding)(struct cmd *, int);
    int      (*check)(struct args *);
    enum cmd_retval (*execc)(struct cmd *, struct cmd_q *);
};

```

As you can see, `cmd_entry` depends on several other types, so we need to work backwards a little bit. If you're going to be lazy and live dangerously, you can get away with it sometimes if you don't care about accessing the data correctly by casting any pointers such as `void *`. But if you're a seasoned C programmer, you know this is fairly dangerous and should be avoided. You can see this type depends on `struct cmd *`, `struct cmd_q *` and `struct args *`. We would ideally want to access these at some point, so it's a good idea to work backwards and implement them one at a time since the rest is just native C types, which Cython understands.

Implementing the `enum` should be by far the simplest:

```

/* Command return values. */
enum cmd_retval {
    CMD_RETURN_ERROR = -1,
    CMD_RETURN_NORMAL = 0,
    CMD_RETURN_WAIT,
    CMD_RETURN_STOP
};

```

Then, turn it into the following:

```

cdef enum cmd_retval:
    CMD_RETURN_ERROR = -1
    CMD_RETURN_NORMAL = 0
    CMD_RETURN_WAIT = 1
    CMD_RETURN_STOP = 2

```

Now that we have the return value for the `exec` hook, we need to look at `struct cmd` next and implement it:

```
struct cmd {
    const struct cmd_entry *entry;
    struct args *args;

    char *file;
    u_int line;

    TAILQ_ENTRY(cmd) qentry;
};
```

Take a look at `TAILQ_ENTRY`. This is simply a preprocessor macro that is a BSD `libc` extension to turn any type into its own linked list. We can ignore this:

```
cdef struct cmd:
    cmd_entry * entry
    args * aargs
    char * file
    int line
```

Note that this struct depends on the `struct cmd_entry` and `struct args` definitions, which we haven't implemented yet. Don't worry about this yet; just put them in for now. Next, let's implement `struct args` since it's simple:

```
/* Parsed arguments. */
struct args {
    bitstr_t *flags;
    char *values[SCHAR_MAX];

    int argc;
    char **argv;
};
```

Note that it uses `bitstr_t` and a variable-length array list. I choose to ignore `bitstr_t` because I think it's a system-dependent header that is fairly tricky to implement. Let's simply cast these as `char *` and `char **` to get things working:

```
cdef struct args:
    char * flags
    char **values
    int argc
    char **argv
```

Now that the `args` structure is Cythonized, let's implement `struct cmd_q`, which is a little more tricky:

```

/* Command queue. */
struct cmd_q {
    int      references;
    int      dead;

    struct client    *client;
    int      client_exit;

    struct cmd_q_items    queue;
    struct cmd_q_item    *item;
    struct cmd          *cmd;

    time_t      time;
    u_int      number;

    void      (*emptyfn)(struct cmd_q *);
    void      *data;

    struct msg_command_data    *msgdata;

    TAILQ_ENTRY(cmd_q)      waitentry;
};

```

There are quite a few more structs that this depends on, but we will not see them here. Let's try and cast these for now; for example, `struct client *`. We can cast this as `void *` and then cast `struct cmd_q_items` simply as `int` even though it isn't correct. As long as we are not going to try and access these fields, we will be okay. But remember that if we were to use Cython `sizeof`, we could run into memory corruption with different sizes allocated by C and by Cython. We can work down the other types such as `struct cmd_q_item *` and cast them as `void *` again. Finally, we come to `time_t` where we can re-use `libc.stdlib cimport time` from Cython. This is a really good exercise for implementing Cython declarations for C applications; it really exercises your code analysis. When going through really long structures, remember that we can get things going by just casting them as `void`; be careful about the struct alignment and typing if you care about the data types in your Cython API.

```

cdef struct cmd_q:
    int references
    int dead
    void * client
    int client_exit
    int queue

```



```
void * item
cmd * cmd
int time
int number
void (*emptyfn) (cmd_q *)
void * msgdata
```

That was a fairly deep dive into a lot of project-specific internals, but I hope you get the idea – we really didn't do anything terribly scary. We even cheated and casted things that we really don't care about. With all these auxiliary types implemented, we can finally implement the type we care about; namely, `struct cmd_entry`:

```
cdef struct cmd_entry:
    char * name
    char * alias
    char * args_template
    int args_lower
    int args_upper
    char * usage
    int flags
    void (*keybinding) (cmd *, int)
    int (*check) (args *)
    cmd_retval (*execc) (cmd *, cmd_q *)
```

With this `cmdpython.pxd` file, we can now implement our Tmux command!

Implementing a Tmux command

One caveat with Cython is that we cannot statically initialize structs like we can in C, so we need to make a hook so that we can initialize `cmd_entry` on Python startup:

```
cimport cmdpython

cdef public cmd_entry cmd_entry_python
```

With this, we now have a public declaration of `cmd_entry_python` which we will initialize in a startup hook as follows:

```
cdef public void tmux_init_cython () with gil:
    cmd_entry_python.name = "python"
    cmd_entry_python.alias = "py"
    cmd_entry_python.args_template = ""
    cmd_entry_python.args_lower = 0
    cmd_entry_python.args_upper = 0
```

```

cmd_entry_python.usage = "python usage..."
cmd_entry_python.flags = 0
#cmd_entry_python.key_binding = NULL
#cmd_entry_python.check = NULL
cmd_entry_python.execc = python_exec

```

Remember that because we declared this in the top level, we know it's on the heap and we don't need to declare any memory to the structure, which is very handy for us. You've seen struct access before; the function suite should look familiar. But let me draw attention to a few things here:

- We declared `public` to make sure we can call it.
- The execution hook is simply a `cdef` Cython function.
- Finally, you might notice the `gil`. I will explain what this is used for in *Chapter 5, Advanced Cython*.


Now let's see a simple execution hook:

```

cdef cmd_retval python_exec (cmd * cmd, cmd_q * cmdq) with gil:
    cdef char * message = "Inside your python command inside tmux!!!"
    log_debug (message)
    return CMD_RETURN_NORMAL;

```

There is not much left to do to hook this into Tmux now. It simply needs to be added to `cmd_table` and the startup hook needs to be added to the server initialization.

 Note that I added something in the `log_debug` function to the PXD; if you look into Tmux, this is a `VA_ARGS` function. Cython doesn't understand these yet, but we can hack it just to get it going by simply casting it as a function that takes a string. As long as we don't try and use it like any `printf`, we should be fine.

Hooking everything together

We now have to fiddle with Tmux just a tiny bit more, but it's fairly painless, and once we are done we are free to be creative. Fundamentally, we should call the `cmd_entry` initialization hook in `server.c` just before we forget about it:

```

#ifdef HAVE_PYTHON
    Py_InitializeEx (0);
    tmux_init_cython ();
#endif

```

```
server_loop();

#ifdef HAVE_PYTHON
    Py_Finalize ();
#endif
```

Now that this is done, we need to make sure we add the `cmd_entry_python` extern declaration to `tmux.h`:

```
extern const struct cmd_entry cmd_wait_for_entry;
#ifdef HAVE_PYTHON
# include "cmdpython.h"
#endif
```

Finally, add this to `cmd_table`:

```
const struct cmd_entry *cmd_table[] = {
    &cmd_attach_session_entry,
    &cmd_bind_key_entry,
    &cmd_break_pane_entry,
    ...
    &cmd_wait_for_entry,
    &cmd_entry_python,
    NULL
};
```

Now that this is done, I think we're good to go; let's test out this baby. Compile Tmux with the following:

```
$ ./configure --enable-python
$ make
$ ./tmux -vvv
$ tmux: C-b :python
$ tmux: exit
```

We can look into `tmux-server-*.log` to see our debug message:

```
complete key ^M 0xd
cmdq 0xbb38f0: python (client 8)
Inside your python command inside tmux!!!
keys are 1 (e)
```

I hope you can now see how easily you can extend this to do something of your own choosing, such as using Python libraries to call directly into your music player, and it would all be integrated with Tmux.

Compiling pure Python code

Another use for Cython is to compile Python code; for example, if we go back to the primes example, we can do the following:

```
philips-macbook:primes redbrain$ cython pyprimes.py -embed
philips-macbook:primes redbrain$ gcc -g -O2 pyprimes.c -o pyprimes
`python-config --includes -libs`
```

Then, we can compare the three different versions of the same program: the fully Cythoned version, the pure Python version, and the Cython-compiled pure Python version:

```
philips-macbook:primes redbrain$ time ./primes
real    0m0.050s
user    0m0.035s
sys     0m0.013s
```

The fully Cython version runs the fastest!

```
philips-macbook:primes redbrain$ time ./pyprimes
real    0m0.139s
user    0m0.122s
sys     0m0.013s
```

The compiled pure Python version runs considerably faster than the pure Python version:

```
philips-macbook:primes redbrain$ time python pyprimes.py
real    0m0.184s
user    0m0.165s
sys     0m0.016s
```

Finally, the pure Python version runs the slowest. But I think it just draws attention to how well Cython can give you some dynamic language optimizations. However, in the real world, if you had several files and needed to compile them correctly, it would not work perfectly. So, if you have a flat import of `foo1.py` and `foo2.py`, you can Cython-compile both and choose the main one by specifying the `-embed` option and then compiling and linking them together. But if you have a module import, you should compile it as a `.so` library and install it via Python `distutils` so that the `.so` Python module can be picked up correctly outside of the working directory of the shared library.

Summary

This chapter was a deep dive into using Cython in a real-world application that many people use, and you really had to worry about very little C. Consider if this .pxd file was added to Tmux; anyone could then write Tmux commands in Python very easily! If you then look at the memory-management hooks, why not wrap all C types into Python classes and reuse the Python garbage collector for memory management. There are so many possibilities—just think outside the box.

4

Debugging Cython

One side effect of using Cython is that it can be fairly tricky to debug your extended applications using tools like **GNU Project Debugger (GDB)**, but that's not to say you cannot do it. You have several choices on what you can do. First and foremost, I must stress on the point that a good practice is to make sure your interfaces between the C and Python code are kept as simple as possible so that what's going on is much clearer. As for me, though I've been developing with C for a long time in large projects, I still am not that much of a user of GDB.

Using GDB on your code

Debugging tools aren't my biggest priority; my GDB sessions are mostly for thread analysis and backtraces. These days, I don't generally make full use of GDB unless there's something really specific in a new project that I've joined where I need to step through the code to understand it a little more in depth. Since Cython is purely C code, you can use GDB over your compiled code if you compile with `-g` to get the debug information from the GNU Collection Compiler (GCC). However, this isn't really that helpful unless you're a Python internals developer. Cython distributions come with **cygdb**, a Cython wrapper for GDB.



If you are on Mac OS X, you will require a GDB version greater than or equal to 0.7, since this was when Python support was added to GDB. By default, you will now have:

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1824) (Wed Feb 6 22:51:23 UTC 2013)
```

I found getting the latest GDB on my MacBook to be painful and full of errors because Xcode wants to control it all. So, I use VirtualBox to have a virtual machine of Ubuntu 12.04 with a bridged networking. I then use Git to synchronize my code and the latest GDB from Ubuntu to do any debugging, having an ssh connection open in another shell and using a GNU Screen or Tmux as a multiplexer.

Running cygdb

Cygdb is mostly a wrapper over GDB (it invokes GDB); it is used before you can debug any native code unlike dynamic language runtimes, such as Python and Java, where you need to generate the debug information. Note that this also generates the `cycode.c` output and the `-gdb` flag will generate the necessary debug information as follows:

```
redbrain@ubuntu-laptop:~/cython-book/chapter4/gdb1$ cython --gdb cycode.pyx
```

Before you start debugging on Ubuntu, you need to install the Python debug information package and GDB, as it is not installed with `build-essential`. To install these, run:

```
redbrain@ubuntu-laptop:~$ sudo apt-get install gdb build-essential cython python-dbg
```

Now that you have GDB and the debug information generated, you can start the Cython debugger with:

```
redbrain@ubuntu-laptop:~/cython-book/chapter4/gdb1$ cygdb . --args python-dbg main.py
```

Once you're familiar with GDB, you can simply use all of the normal `gdb` commands. However, the whole point of `cygdb` is that we can use the Cython commands, which we will see in use here with an explanation:

```
(gdb) cy break
__init__          cycode.foobar.__init__  cycode.foobar.print_me
cycode.func      func                    print_me
```

If you tab autocomplete `cy break`, you will see the list of symbols to which you can set a Cython break point. Next, we need to get the program running and then continue to our break points as follows:

```
(gdb) cy break func
Function "__pyx_pw_6cycode_1func" not defined.
Breakpoint 1 (__pyx_pw_6cycode_1func) pending.
```

Now that we have the break point set, we need to run the program:

```
(gdb) cy run
1  def func (int x):
```

Now that we have hit the declaration of the `func` function, we can continue and do some introspection as follows:

```
(gdb) cy globals
Python globals:
```

```
__builtins__ = <module at remote 0x7ffff7fabb08>
__doc__      = None
__file__     = '/home/redbrain/cython-book/chapter4/gdb1/cycode.so'
__name__     = 'cycode'
__package__  = None
__test__     = {}
foobar      = <classobj at remote 0x7ffff7ee50b8>
func        = <built-in function func>
```

C globals:

The `globals` command will show any of the global identifiers in the scope of the current frame, so we can see the `func` function and the `classobj` `foobar`. We can inspect further by listing the code and step code:

```
(gdb) cy list
1   def func (int x):
2       print x
3       return x + 1
4
```

We can also step the code:

```
(gdb) cy step
1
4   cycode.func (1)

(gdb) cy list
1   #!/usr/bin/python
2   import cycode
3
4   cycode.func (1)
> 5   object = cycode.foobar ()
6   object.print_me ()
```

```
(gdb) cy step
3       return x + 1
```

```
(gdb) cy list
1   def func (int x):
2       print x
```



```
> 3         return x + 1
4
5     class foobar:
6         x = 0
7         def __init__ (self):
```

You can get fairly neat listings even from classes:

```
(gdb) cy list
3         return x + 1
4
5     class foobar:
6         x = 0
7         def __init__ (self):
> 8             self.x = 1
9
10        def print_me (self):
11            print self.x
```

We can even see the backtrace of the current Python state:

```
(gdb) cy bt
#9  0x000000000047b6a0 in <module>() at main.py:6
      6     object.print_me ()
#13 0x00007ffff6a05ea0 in print_me() at /home/redbrain/cython-book/
chapter4/gdb1/cycode.pyx:8
      8         self.x = 1
```

The help can be found by running the following command:

```
(gdb) help cy
```

I think you have got the idea! It's worth playing around and checking the help and trying these for yourself to get the feel of debugging with cygdb. To get a good feel, you really need to practice with GDB and get comfortable with it.

General Cython caveats

There are several caveats worth noting when it comes to Cython while mixing C and the Python code. As for me, I tend to keep the interfaces between the C and Python projects as simple as possible so that it's clear when and where something wrong is going on.

Type checking

You may have noticed that in previous code examples, we were able to cast the `void *` pointer from `malloc` to our extension types using `malloc`. Cython supports some more advanced type checking, for example:

```
char * buf = <char *> malloc (sizeof (...))
```

In basic type casting, Cython supports `<type?>` for type checking for example:

```
char * buf = <char *?> malloc (...)
```

This will do some type checking and will throw an error if the type being cast is not a subclass of `char *`. So, in this case, it will pass; however, if you were to do the following:

```
cdef class A:
    pass
cdef class B (A):
    pass

def myfunc ():
    cdef A class1 = A ()
    cdef B class2 = B ()
    cdef B x = <B?> class1
```

This will return an error (at runtime):

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    myfunc ()
  File "cycode.pyx", line 12, in cycode.myfunc (cycode.c:714)
    cdef B x = <B?> class1
TypeError: Cannot convert cycode.A to cycode.B
```

So, this could add some more type safety to your Cython APIs.

No * operator

In Cython, we don't have any pointer dereference operator; for example, if you are passing a C array, you can use pointer arithmetic like the following to print it:

```
int * ptr = array;
int i;
for (i = 0; i < len; ++i)
    printf ("%i\n", *ptr++);
```

In Cython, we have to be a little more verbose or explicit:

```
cdef int i
cdef int * ptr = array
for i in range (len):
    print ptr [0]
    ptr = ptr + 1
```

There is nothing really fancy here; you simply have to use `x[0]` if you want to dereference `int *x`.

Python exceptions in C

Another topic to look at is what happens if your Cython code propagates an exception to your C code. In the next chapter, we will cover how C++ native exceptions interact with Python, but we do not have this in C. Consider the following code:

```
cdef public void myfunc ():
    raise Exception ("Raising an exception!")
```

This simply raises an exception back to C and gives the following:

```
10-4-5-52:exceptions redbrain$ ./test
Exception Exception: Exception('Raising an exception!') in 'cycode.
myfunc' ignored
Away doing something else now...
```

As you can see, a warning was printed and no exception handling occurred, so the program continues on to something else. This is because the plain `cdef` functions that do not return Python objects have no way for exceptions to be handled, and thus a simple warning message is printed. If we want to control the behavior for C programs, we need to declare the exception on the Cython function prototype.

There are three forms for doing this. Firstly, we can do:

```
cdef int myfunc () except -1:
    cdef int retval = -1
    ...
    return retval
```

This makes the function throw an exception on the function returning `-1` at any point. This also causes the exception to be propagated to the caller; so, in Cython, we can do the following:

```
cdef public void run ():
    try:
```

```

myfunc ()
    somethingElse ()
except Exception:
    print "Something wrong"

```

You can also use the "maybe" exception (as I would like to think of it), which looks like:

```

cdef int myfunc () except ? -1:
    cdef int retval = -1
    ...
    return retval

```

This means that it may or may not be an error, and Cython generates a call to `PyErr_Occurred` to perform verification from the C API. Lastly, we can use the wildcard:

```

cdef int myfunc () except *:

```

This then makes it always call `PyErr_Occurred`, which you can check via `PyErr_PrintEx` or others at <http://docs.python.org/2/c-api/exceptions.html>.

Please note that the function pointer declarations can also handle this in their prototype. Just make sure that the return type matches the exception type, which must be an enum, float, pointer-type, or constant expression; if not, you will get a confusing compilation error.

For loops on C types

Cython has more support for C style `for` loops and can also perform further optimizations on the `range` function depending on how the iterator is declared. Generally in Python, you simply do the following:

```

for i in iterable_type: ...

```

This is fine on PyObjects since they understand iterators, but C types do not have any of this abstraction and you need to do pointer arithmetic on your array types to access indexes. So, for example, first we can do the following with the `range` function:

```

cdef void myfunc (int length, int * array)
    cdef int i
    for i in range (length):
        print array [i]

```

When the `range` function is used on C types, such as the following example that uses `cdef int i`, it is optimized for real C array access. There are several other forms we can use. We could translate the loop into the following:

```
cdef int i
for i in array [:length]: print i
```

This looks a lot more like a normal Python `for` loop performing the iteration assigning `i`, the index data. There is also one last form that Cython introduces using the `for .. from` syntax. This looks like a real `for` loop from C, and we could now write:

```
def myfunc (int length, int * array):
    cdef int i
    for i from 0 <= i < length;
        print array [i]
```

We can also introduce the step size:

```
for i from 0 <= i < length by 2:
    print array [i]
```

These extra `for` loop constructs are particularly useful when working a lot with C types because they do not understand extra Python constructs.

Bool type

When you try and use `bool` in Cython, you will get:

```
cycode.pyx:2:9: 'bool' is not a type identifier
```

So, you need to use:

```
from libcpp cimport bool
```

Then, when you compile, you might get the following:

```
cycode.c: In function '__pyx_pf_6cycode_run':
cycode.c:642: error: 'bool' undeclared (first use in this function)
cycode.c:642: error: (Each undeclared identifier is reported only once
cycode.c:642: error: for each function it appears in.)
cycode.c:642: error: expected ';' before '__pyx_v_mybool'
cycode.c:657: error: '__pyx_v_mybool' undeclared (first use in this
function)
```

You need to make sure you're compiling with a C++ compiler as `bool` is a native type.

No C const

You may have noticed that I've not used `const` anywhere when it comes to Cython code. Cython doesn't understand the `const` keyword from Cython 0.18-pre, but we can work around this with the following:

```
cdef extern from *:
    ctypedef char* const_char_ptr "const char"
```

Now we can use the `const` keyword like this:

```
cdef public void foo_c(const_char_ptr s):
    ...
```

If you're using Cython greater than or equal to 0.18, you can use `const` just as you would from C.

Multiple Cython files

Cython does not handle multiple `.pyx` files. So, Cython has another keyword and convention: `.pxi`. This is an extra include file that works just like C includes. All the other Cython files get pulled into one file that makes one huge Cython compilation. For this, you need to do the following:

```
include "myothercythonfile.pxi"
```

Initializing struct

When declaring `struct`, you cannot do normal C initialization such as:

```
struct myStruct {
    int x;
    char * y;
}
struct myStruct x = { 2, "bla" };
```

You need to do:

```
cdef myStruct x:
    x.x = 2
    x.y = "bla"
```

So, you manually specify the fields more verbosely.

Calling into pure Python modules

You can always call into pure Python code (non-Cythoned), but you should always beware and use Python `distutils` to make sure the module is installed correctly outside of the development environment.

Keeping call stacks small and pure

When passing data between Python and C, try to keep your wrapper functions as small as possible.

Summary

Overall, debugging in my opinion is a fairly per-person preference based on how you tend to manage your workflow. So long as you keep your interfaces between C and Python, your code is very simple to follow and debug. In the next chapter, we will concentrate on how you can get more advanced usage from Cython and how we can use C++ and C++ exceptions with Python exceptions, templates, and classes.

5

Advanced Cython

So far with Cython we have only been using C for all our examples and our code. But with Cython we can also target C++ and work with most of its constructs. With better support in each of the new releases, you can tell that this is getting some major development attention since most people use `python: :boost` to embed Python in C++ code, which has a plethora of dependencies. So, if you're an avid user of C++, I would recommend checking up Cython release notes regularly to see more support features. Since updating to new versions of tools such as Cython and Bison doesn't really affect your code, it just makes your project more efficient and better with each release.

C++ constructs

I will go over a simple example of each of the main C++ constructs to help you translate and use them to get an idea of how native this really feels in Cython!

Namespaces

We can handle namespaces by using the Cython `namespace` keyword. So, for example, the namespace defined in a C++ header could be translated into the following:

```
#ifndef __MY_HEADER_H__
#define __MY_HEADER_H__

namespace mynamespace {
...
}

#endif //__MY_HEADER_H__
```

You will wrap this with the `cdef extern` declaration:

```
cdef extern from "header.h" namespace "mynamespace":
...
```


And you can now address it in Cython as you normally would do for a module:

```
import cythonfile
cythonfile.mynamespace.attribute
```

It really feels like a python module simply by using a namespace.

Classes

I would take a guess that most of your C++ code revolves around using classes. Being an object-oriented language, Cython also handles this in a really simple manner. But remember that Cython will care about the `public` attributes only, since these are the only attributes a callee can access due to the encapsulation of private and protected methods.

```
#ifndef __MY_HEADER_H__
#define __MY_HEADER_H__

namespace mynamespace {
    void myFunc (void);

    class myClass {
    public:
        int x;
        void printMe (void);
    };
}

#endif //__MY_HEADER_H__
```

You then just need to care about your class' public attributes, which would become:

```
cdef extern from "myheader.h" namespace "mynamespace":
    void myFunc ()
    cppclass myClass:
        int x
        void printMe ()
```

We only care about the public members of C++ classes because we can only access the public attributes in the caller sense. Now you can work with these just as if they were `cdef` structs. Just use the `.` operator as before and you can access all the necessary attributes.

C++ new keyword and allocation

Cython understands the new keyword from C++; so, consider you have a C++ class:

```
class Car {
    int doors;
    int wheels;
public:
    Car ();
    ~Car ();
    void printCar (void);
    void setWheels (int x) { wheels = x; };
    void setDoors (int x) { doors = x; };
};
```

And it is defined in Cython as:

```
cdef extern from "cppcode.h" namespace "mynamespace":
    cppclass Car:
        Car ()
        void printCar ()
        void setWheels (int)
        void setDoors (int)
```

Note that we do not declare the ~Car destructor since you never call this directly. It's not really a public member, hence why we never call it directly. We then instantiate the raw C++ class in Cython code using new:

```
cdef Car * c = new Car ()
```

You can then go and use del to delete the object at any time:

```
del c
```

You will see that the destructor is called as you would expect:

```
10-4-5-52:cppalloc redbrain$ ./test
Car constructor
Car has 3 doors and 4 wheels
Car destructor
```

We can also declare a stack-allocated object, but it must only have a default constructor such as:

```
cdef Car c
```

There is no way of passing arguments with this syntax in Cython. But note you cannot use del on this instance else you will get the following error:

```
cpycode.pyx:13:6: Deletion of non-heap C++ object
```

Exceptions

In my opinion, with C++ exception handling, you can get a sense of how integrated Cython can feel within C++ code; Python just figures it out really well! If any exception is thrown such as memory allocations, Python will handle these and translate them into more useful errors, and you still get the normal little `ex.what ()` message to your `stdout` stream if you desire. Python will also understand if these are caught or not and whether they are handled as required. This table gives you an idea of what Python exceptions will map to normal C++ ones:

C++	Python
<code>bad_alloc</code>	<code>MemoryError</code>
<code>bad_cast</code>	<code>TypeError</code>
<code>domain_error</code>	<code>ValueError</code>
<code>invalid_argument</code>	<code>ValueError</code>
<code>ios_base::failure</code>	<code>IOError</code>
<code>out_of_range</code>	<code>IndexError</code>
<code>overflow_error</code>	<code>OverflowError</code>
<code>range_error</code>	<code>ArithmeticError</code>
<code>underflow_error</code>	<code>ArithmeticError</code>
All other exceptions	<code>RuntimeError</code>

Take, for instance, this C++ code that will simply throw an exception when the `myFunc` function is called. Firstly, we define an exception with:

```
namespace mynamespace {
    class mycppexcept: public std::exception {
        virtual const char * what () const throw () {
            return "C++ exception happened";
        }
    };

    void myFunc (void) throw (mycppexcept);
}
```

Now we write the function to throw the exception:

```
void mynamespace::myFunc (void) throw (mynamespace::mycppexcept) {
    mynamespace::mycppexcept ex;
    cout << "About to throw an exception!" << endl;
    throw ex;
}
```

We can call this in Cython with simply:

```
cdef extern from "myheader.h" namespace "mynamespace":
    void myFunc () except +RuntimeError
```

And when we run the function, we get the following output:

```
>>> import cpycode
About to throw an exception!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "cpycode.pyx", line 3, in init cpycode (cpycode.cpp:763)
    myFunc ()
RuntimeError: C++ exception happened
>>> ^D
```

Also, if you want, you can catch the C++ exception in your Python code using:

```
try:
    ...
except RuntimeError:
    ...
```

Notice we told Cython to cast any exceptions to `RuntimeError`. This is important to make sure you understand where and which interfaces can throw exceptions, since unhandled exceptions look really ugly and cause pain. And Cython cannot really assume much, since compilers won't throw errors on unhandled exceptions in C++; so, you will get the following as we didn't declare any exception handling on the function:

```
Philips-MacBook:cppexceptions redbrain$ python
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on
darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import cpycode
About to throw an exception!
Segmentation fault: 11
```

Bool type

As seen in the previous chapter, to use the native `bool` type from C++, you need to firstly import:

```
from libcpp cimport bool
```

Then, you can use `bool` as a normal `cdef`. If you want to use the pure PyObject `bool` type, you need to import:

```
from cpython cimport bool
```

You can then assign them with the normal `true` or `false` values.

Overloading

Overloading is very simple. Since Python supports this natively by default, just list more functions like:

```
cdef foobar (int)
cdef foobar (int, int)
...
```

Cython understands that we are in C++ mode and can handle all the type conversion as normal. It's interesting that it can also handle an operator overload easily since it is just another hook! For example, let's take the `Car` class again and perform some operator overriding such as:

```
namespace mynamespace {
    class Car {
        int doors;
        int wheels;
    public:
        Car ();
        ~Car ();
        Car * operator+(Car *);
        void printCar (void);
        void setWheels (int x) { wheels = x; };
        void setDoors (int x) { doors = x; };
    };
};
```

Remember to add these operator-overloading class members to your Cythonized class; otherwise, your Cython will throw the following error:

```
Invalid operand types for '+' (Car *, Car *)
```

The Cython declaration of the operator overload looks as you might expect:

```
cdef extern from "cppcode.h" namespace "mynamespace":
    cppclass Car:
        Car ()
        Car * operator+ (Car *)
        void printCar ()
        void setWheels (int)
        void setDoors (int)
```

Now you can do the following:

```
cdef Car * ccc = c[0] + cc
ccc.printCar ()
```

This will then give us the following output on the command line:

```
10-4-5-52:cppoverloading redbrain$ ./test
Car constructor
Car constructor
Car has 3 doors and 4 wheels
Car has 6 doors and 8 wheels
inside operator +
Car constructor
Car has 9 doors and 12 wheels
```

Then, everything is handled as you would expect if you try to add classes of the same type. You can really feel the inspiration of Python when you see these hooks in action and the ideas of being more dynamic at work.

Templates

Templates are supported, though not 100 percent, and are the simplest way to debug compilation errors; I wrote a `LinkedList` template class in `chapter5/cpptemplates` which you can try. But in essence, you can simply follow on how you implement a class in Cython where you declare it with the `cppclass name [T]` syntax:

```
cppclass LinkedList [T]:
    LinkedList ()
    void append (T)
    int getLength ()
    ...
```

Now you can access the template type with declaration `T`.

Static class member attribute

Sometimes in classes, it's useful to have a static attribute such as:

```
namespace mynamespace {
    class myClass {
    public:
        static void myStaticMethod (void);
    };
}
```

In Cython, there is no support for this via a `static` keyword, but what you can do is tie this function to a namespace so that it would become:

```
cdef extern from "header.h" namespace "mynamespace::myClass":
    void myStaticMethod ()
```

Now you simply call that method.

Caveat on C++ usage

There are several more caveats on C++ usage to keep in the back of your mind when using it.

Calling in C and C++ functions

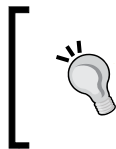
When you write a code to call in a C++ function from C, you need to wrap the prototypes in the following:

```
extern "C" { ... }
```

This allows you to call C++ prototypes because C won't understand a C++ class. With Cython, if you are telling your C output to call in C++ functions, you need to be careful about which compiler you are using or you need to write a new header to implement the minimal wrapper functions required to make the C++ calls.

Namespaces

Cython seems to generally require a namespace to keep things nested, which you are probably doing in your C++ code already. Making PXD on non-namespaced code seems to make new declarations, meaning that you will get linking errors due to multiple symbols. C++ support looks really good from these templates, and more metaprogramming idioms can be difficult to express in Cython. When polymorphism comes into play, it can be difficult to track down compilation errors. I would stress keeping your interfaces as simple as possible to perform debugging and being more dynamic!



Remember when using Cython to generate the C++ you need to specify `-cplus` so it will default the `cythonfile.cpp` output. Pay attention to the extensions; I prefer to use `.cc` for my C++ code, so just be careful with your build system.

Python distutils

As usual, we can also use Python `distutils`, but you will need to specify the language so that the auxiliary C++ code required will be compiled by the correct compiler:

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

setup (ext_modules = cythonize(
    "mycython.pyx",
    sources = ["mysource.cc"],
    language = "c++",
))
```

Now you can compile your C++ code to your Python module.

Python threading and GIL

GIL stands for **Global Interpreter Lock**. What this means is when you link your program against `libpython.so` and use it, you really have the entire Python interpreter in your code. The reason this exists is to make concurrent applications really easy. In Python, you can have two threads reading/writing to the same location (variable) Python automatically handles all of this for you; unlike say in Java, where you need to specify that everything is under the GIL in Python. There are two things to consider when talking about the GIL and what it does: instruction atomicity and read/write lock.

Atomic instructions

Remember that Cython generates the C code necessary to make it look just like any Python module that you can import. So what's happening under the hood is that it will generate all the code to acquire lock on the GIL so that it can manipulate Python objects at runtime. Let's consider two types of execution. Firstly, you have the C stack where it executes atomically as you would expect; it doesn't care about synchronization between threads – that's left up to the programmer. The other is Python where it's doing all of this synchronization for us. When you embed Python into your application manually using `Py_Initialize`, this is under the C execution. When it comes to calling something such as `import sys` and `sys.uname` in Cython code that is called from C, the Python GIL schedules and blocks multiple threads from calling this at the same time to be safe. This makes writing multithreaded Python code extremely safe. So, any errors from writing to the same location at the same time can happen and be handled correctly instead of having to use mutexes on critical sections in C. Anything that runs Python calls, such as folding types, iterators, and type conversions, has to go under the GIL and is schedule-based against different threads to preserve consistency.

But in the end, it runs direct C code, so this means your C executes just as you would expect any C program to. It's only when called directly that Python code has the potential to be blocked by the GIL or worsen the objects being dereferenced. This is by far the most common problem with people writing their own C module from scratch, as you really do need good knowledge of garbage collection and object lifetime within your application. Because Cython compiles to C/C++, this is where a lot of its optimizations come from because you're not reliant on the GIL for all code execution.

Read/write lock

Read/write lock is great because no matter what, it is pretty rare for you in Python to really need to care about semaphores or mutexes on data. The worst that can happen is you get into an inconsistent state, but you won't crash like you might in C. Any read/write operation to the global dictionary is handled the way you would expect in Python by blocking threads when needed.

Cython keywords

Okay, so how does this affect you and, more importantly, your code? It is important to know what way your code should and/or will execute in a concurrent manner. Without an understanding of this, you will be debugging for hours and not know what's going on. There are times when the GIL gets in the way and can cause issues by blocking execution of your C code from Python or vice versa. In Python, this is fairly confusing until you spend time on IRC with some developers to help you, but Cython lets us control the GIL with the `gil` and `nogil` keywords:

Cython	Python
With <code>gil</code>	<code>PyGILState_Ensure ()</code>
With <code>nogil</code>	<code>PyGILState_Release (state)</code>

It's fairly abstract to keep talking about Python in this way. Although not 100 percent correct in all the details, it should give you a gentle introduction to what I am talking about. I find it's easier to think of multithreading in Python in terms of blocking and nonblocking execution, since Python does so well at writing concurrently to Python objects. In the next example, we will examine the steps needed to embed a Twisted web server into the messaging server we implemented.

Messaging server revisited

The messaging server is an example of something that would be highly concurrent; let's say we want to embed a web server into this to show the list of clients that are connected to the server. We could easily reuse Python Twisted. If you look at Twisted, you can see how easily you can have a fullblown web container in about eight lines of code.

Let's do it! We know from the messaging server that the callbacks all call into the roster, so we can iterate over the roster dictionary to get online clients and simply return some HTML to display this in your browser.

One thing with embedding web servers is that they start a lot of threads, meaning they really need to be started via the main thread. If we were to start our `libevent` engine via a Python thread, we could move on and start the web server just like the example. So, if you look at `pyserver.pyx`, we could do:

```
class RunCServer (threading.Thread):
    def __init__ (self, port):
        self.port = port
        threading.Thread.__init__ (self)
```

```
def run (self):
    cdef int cport = self.port
    with nogil:
        init_server (cport)
```

Then, as you would expect, we can start the thread by running this::

```
# start libvent server
cs = RunCServer (port)
cs.start ()
```

Notice that I specified `with nogil`. We are calling into our C engine here, and it will not care about Python. Otherwise this will block, and our web server won't start until after it exits from the event loop. But really, our C code doesn't need the GIL since we are only using pure C types and data there. Once the `libevent` socket server is running asynchronously, we can then move on to starting our web server.

```
# start webserver
webserver.WebServer (webport, roster)
cs.join ()
```

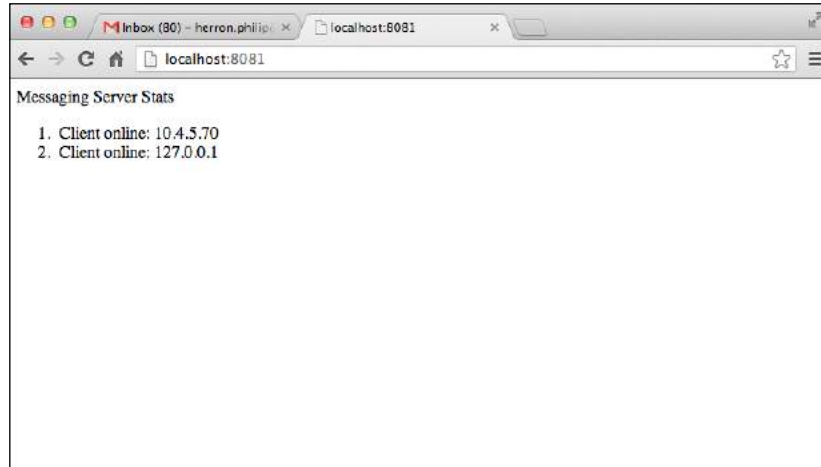
We let the web server start here and block as the Twisted web server tutorial shows us, and it reads from `global` to display data. The server now blocks until the kill signal is given, and it will return and join (`wait`) once the event loop cleans up and exists correctly:

```
def WebServer (port, roster):
    global DATA
    DATA = roster
    site = server.Site (Simple())
    reactor.listenTCP (port, site)
    reactor.run ()
```

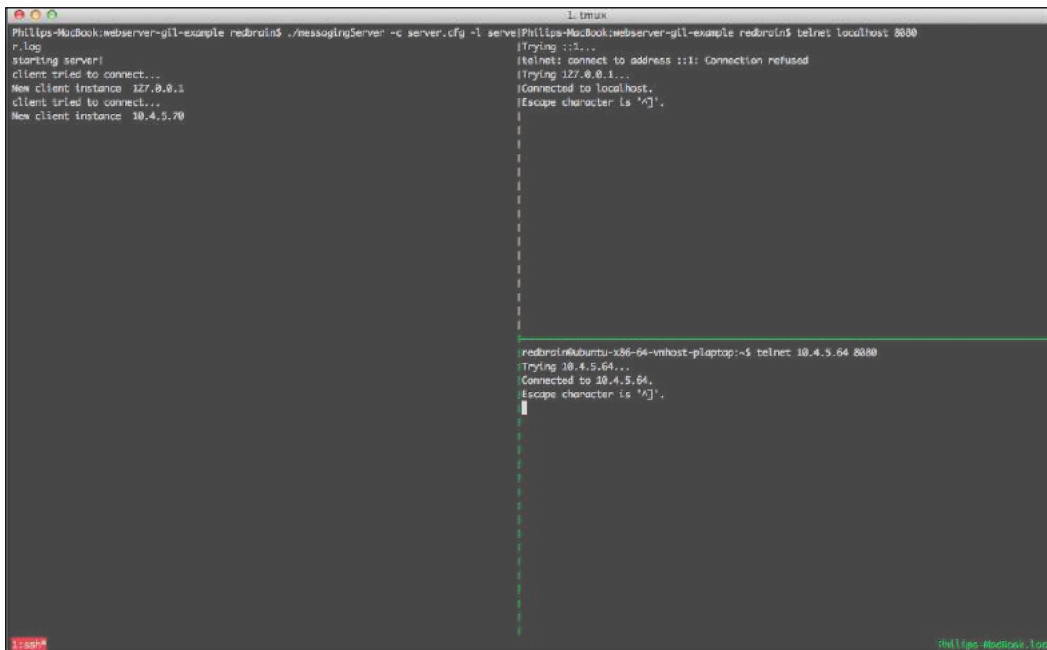
Now we are listening on the specified port in `server.cfg`:

```
[pyserver]
port = 8080
webport = 8081
```

We can see the web page as shown in the following screenshot:



No clients are connected, but when they are, you can see them go online and offline with a refresh. Now this looks incredibly simple! This is shown in the following screenshot:

A screenshot of a terminal window. The top part shows the server starting and logging client connections from 127.0.0.1 and 10.4.5.70. The bottom part shows a telnet session from a remote host (10.4.5.64) connecting to the server on port 8080. The terminal output is as follows:

```
Phillips-MacBook:webserver-gli-example redbruin$ ./messagingServer -c server.cfg -l serve
r.log
starting server!
client tried to connect...
New client instance 127.0.0.1
client tried to connect...
New client instance 10.4.5.70

redbruin@ubuntu-x86_64-vmhost-plettap:~$ telnet 10.4.5.64 8080
Trying 10.4.5.64...
Connected to 10.4.5.64.
Escape character is '^['.
```

There is one fairly huge caveat to remember when using `gil`. In our callbacks, we need to acquire the GIL on each callback before we call any Python code or we will segfault and get really confused. So if you look into each of the `libevent` callbacks, when calling the Cython functions, you have the following:

```
PyGILState_STATE gilstate_save = PyGILState_Ensure();
readcb (client, (char *)data);
PyGILState_Release(gilstate_save);
```

Notice this is also called on the other two callbacks, firstly on the `discb` callback:

```
PyGILState_STATE gilstate_save = PyGILState_Ensure();
discb (client, NULL);
PyGILState_Release(gilstate_save);
```

And finally on the `connect` callback, we must be a little more safe and call it this way:

```
PyGILState_STATE gilstate_save = PyGILState_Ensure();
if (!conncb (NULL, inet_ntoa (client_addr.sin_addr))
    {
...
    }
else
    close (client_fd);
PyGILState_Release(gilstate_save);
```

We have to do this since we executed this with `nogil` from Cython. We need to acquire `gil` before we go back into Python land. You really need to look at something like this with your creativity cap on and imagine what you could do with this. For example, you could use this as ways to capture data and use the Twisted web server to implement an embedded restful server. Maybe even use Python JSON to wrap data into nice objects. But, moreover, it demonstrates how you really can extend a fairly complicated piece of C software with something nice and of a high-level nature using Python libraries. This keeps everything very simple and maintainable instead of trying to do everything from scratch.

More inspiration

Extending C/C++ systems with Python is an attractive idea, but deciding on which way to extend it can be difficult. Here are two ideas which I have used with success.

Messaging server working with SQL

Another example could be serializing data to a database using Python libraries such as SQLite3. But, we could extend our messaging server examples to provide outputs to SQL databases such as SQLite:

```
import sqlite3
sqlconn = None
```

Then, in the initialization, we could:

```
def pyinit_server (port):
    global roster, sqlconn
    sqlconn = sqlite3.connect ('sqlite.db')
```

Finally, in the callbacks:

```
cdef int pyconnect_callback (client *c, char * args):
    global sqlconn
    sqlconn.execute ("SQL....")
```

Now we can successfully output the data into an SQLite database with very little work.

Python IRC notifier

We could write our own messaging server and export it to IRC via the Python IRC module (<https://pypi.python.org/pypi/irc/>).

Then, you can add in code to each of the hooks to push it to a specified IRC server and channel. If you have a bot connected, you could pass it to a queue of messages to send if you thread it off correctly.

Unit testing the native code

Another use of Cython is unit testing the core functionality of shared C libraries. If you maintain a .pxd file (this is all you need really), you could write your own wrapper classes and do scalability testing of data structures with the expressiveness of Python. For example, we could write unit tests for something like `std::map` and `std::vector` such as:

```
from libcpp.vector cimport vector

PASSED = False

cdef vector[int] vect
cdef int i
for i in range(10):
    vect.push_back(i)
```

```
for i in range(10):  
    print vect[i]
```

```
PASSED = True
```

And then write a test for map as follows:

```
from libcpp.map cimport map
```

```
PASSED = False
```

```
cdef map[int,int] mymap
```

```
cdef int i
```

```
for i in range (10):
```

```
    mymap[i] = (i + 1)
```

```
for i in range (10):
```

```
    print mymap[i]
```

```
PASSED = True
```

Then, if we compile them into separate modules, we could simply write a test executor:

```
#!/usr/bin/env python  
print "Cython C++ Unit test executor"
```

```
print "[TEST] std::map"
```

```
import testmap
```

```
assert testmap.PASSED
```

```
print "[PASS]"
```

```
print "[TEST] std::vec"
```

```
import testvec
```

```
assert testvec.PASSED
```

```
print "[PASS]"
```

```
print "Done..."
```

This is really trivial code, but it demonstrates the idea. If you put error handling with plenty of asserts and cause a fatal error, you could have some really nice unit testing against your C/C++ code. We could go further and implement this using Python's native unit testing framework.

Preventing subclassing

If you create an extension type in Cython, something you do not want ever to be subclassed is a `cpp` class wrapped in a Python class. To prevent this, you can do the following:

```
cimport cython

@cython.final
cdef class A: pass

cdef class B (A): pass
```

This annotation will give an error when someone tries to subclass:


```
pycode.pyx:7:5: Base class 'A' of type 'B' is final
```

Note that these annotations only work on the `cdef` or `cpdef` functions and not on normal Python `def` functions.

Cython typing via annotations

Some people may find using the explicit `cdef` and C type syntax awkward. In Cython functions, you can augment the usage of C types. You can use this annotation as follows:

```
@cython.locals(x = cython.int, y = cython.int)
def function ():
    ...
```

[ Argument types are also handled via the `@cython.locals` annotation.]

This will then generate the necessary code instead of explicitly using `cdef` and C types in your code. There are several more annotations you can use so you don't have to use the Cython syntax:

```
@cython.cfunc
@cython.returns(cython.int)
@cython.locals(x = cython.int, y = cython.int)
def function ():
    ...
    return x + y
```


The `cfunc` annotation tells Cython that this is a `cdef` function and the `returns` annotation tells Cython the return type for `cdef`. Remember that if you want to add normal Python functions as part of a `.pxd` file to `cimport`, this will fail. You need to use `cpdef` to achieve this as explained in *Chapter 6, Further Reading*.

Parsing large amounts of data

I want to try and prove how powerful and natively compiled C types are to programmers by showing the difference in parsing large amounts of XML. We can take the geographic data from the government as the test data for this experiment (http://www.epa.gov/enviro/geo_data.html):

Let us look at the size of this XML data:

```
10-4-5-52:bigData redbrain$ ls -liah
total 480184
7849156 drwxr-xr-x  5 redbrain  staff   170B 25 Jul 16:42 ./
5803438 drwxr-xr-x 11 redbrain  staff   374B 25 Jul 16:41 ../
7849208 -rw-r--r--@  1 redbrain  staff  222M  9 Mar 04:27
EPAXMLDownload.xml
7849030 -rw-r--r--@  1 redbrain  staff   12M 25 Jul 16:38
EPAXMLDownload.zip
7849174 -rw-r--r--   1 redbrain  staff   57B 25 Jul 16:42 README
```

It's huge! Before we write programs, we need to understand a little bit about the structure of this data to see what we want to do with it. It contains facility sites' locations with addresses. This seems to be the bulk of the data in here, so let's try and parse it all out with a pure Python XML parser using:

```
from xml.etree import ElementTree as etree
```

The code uses `etree` to parse the XML file via:

```
xmlroot = etree.parse (__xmlFile)
```

Then, we look up the header and facilities via:

```
headers = xmlroot.findall ('Header')
facts = xmlroot.findall ('FacilitySite')
```

Finally, we output them to a file:

```
try:
    fd = open (__output, "wb")
    for i in facts:
        location = ""
        for y in i:
```

```

        if isinstance (y.text, basestring):
            location += y.tag + ": " + y.text + '\n'
        fd.write (location)
    # There is some dodgy unicode character
    # python doesn't like just ignore it
    except UnicodeEncodeError: pass
    except:
        print "Unexpected error:", sys.exc_info()[0]
        raise
    finally:
        if fd: fd.close ()

```

We then time the execution as follows:

```

10-4-5-52:bigData redbrain$ time python pyparse.py
USEPA Geospatial DataEnvironmental Protection AgencyUSEPA Geospatial
DataThis XML file was produced by US EPA and contains data specifying the
locations of EPA regulated facilities or cleanups that are being provided
by EPA for use by commercial mapping services and others with an interest
in using this information. Updates to this file are produced on a regular
basis by EPA and those updates as well as documentation describing the
contents of the file can be found at URL:http://www.epa.gov/enviro
MAR-08-2013
[INFO] Number of Facilties 118421
[INFO] Dumping facilities to xmlout.dat

real    2m21.936s
user    1m58.260s
sys     0m9.5800s

```

This is quite long but let's compare it using a different XML implementation, Python `lxml`. It's a different library implemented using Cython but implements the same library as the previous pure Python XML parser.

```
10-4-5-52:bigData redbrain$ sudo pip install lxml
```

We can simply drop the replacement import in to:

```
from lxml import etree
```

The code stays the same, but the execution time is dramatically reduced (compile the Cython version by running `make` and the `cpyparse` binary is created from the same code with just a different import).

```
10-4-5-52:bigData redbrain$ time ./cpyparse
USEPA Geospatial DataEnvironmental Protection AgencyUSEPA Geospatial
DataThis XML file was produced by US EPA and contains data specifying the
locations of EPA regulated facilities or cleanups that are being provided
by EPA for use by commercial mapping services and others with an interest
in using this information. Updates to this file are produced on a regular
basis by EPA and those updates as well as documentation describing the
contents of the file can be found at URL:http://www.epa.gov/enviro
MAR-08-2013
[INFO] Number of Facilties 118421
[INFO] Dumping facilities to xmlout.dat

real    0m7.874s
user    0m5.307s
sys     0m1.839s
```

You can really see the power of using native code when you make just a little effort. And to be finally assured that the code is the same, let's MD5 sum `xmlout.dat` that we created:

```
10-4-5-52:bigData redbrain$ md5 xmlout.dat xmlout.dat.cython
MD5 (xmlout.dat.python) = c2103a2252042f143489216b9c238283
MD5 (xmlout.dat.cython) = c2103a2252042f143489216b9c238283
```

So, you can see that the outputs are exactly the same just so we know that no funny business is going on. It's scary how much faster this can make your XML parsing and if we calculate the speed increase rate, it is approximately 17.75 times faster; but don't take my word for it, try running it yourself. My MacBook has a solid state disk and has a 4 GB RAM with a 2 GHz Core 2 Duo.

Summary

I think this chapter covers quite a lot of the Python internals that you should care about. If you have ever tackled writing a native Python module for a multithreaded system, you have undoubtedly come across a lot of issues; when you get comfortable, Cython really does make it so much simpler. If you use the `gil` keywords, we can handle this in a more native way. The other major problem people have is the garbage collector and making sure your reference counting is correct. And really, unless you are a Python developer writing a module from scratch in C, this becomes a major task that needs a lot of expertise and time dedicated to maintenance. With Cython, if you know C and Python, you are sorted.

Chapter 6, Further Reading, is the final chapter, and I want to round out the book with some final caveats and usages with Cython. I will talk about how you can use Cython with Python 3 and PyPy. I We will also visit and look at Python projects such as AutoPxd for Cython. Finally, I want to discuss some similar projects such as Numba and SWIG versus Cython.

6

Further Reading

So far in this book, we have looked into both the basic and advanced topics of using Cython. But it does not stop here; there are further topics that you can explore. Consider other implementations of Python such as PyPy or making it work with Python 3. Other topics we will discuss in this chapter are OpenMP support and type casting and object initialization in Cython.

Keyword `cpdef`

Currently, we have seen two different function declarations in Cython, `def` and `cdef`, to define functions. There is one more declaration: `cpdef`. `def` is a Python-only function, so it is only callable from Python or Cython code blocks; calling from C does not work. `cdef` is the opposite; this means that it's callable from C and not from Python. For example, if we create a function such as:

```
cpdef public test (int x):  
    ...  
    return 1
```

This will generate the following function prototype:

```
__PYX_EXTERN_C DL_IMPORT(PyObject) *test(int, int __pyx_skip_  
dispatch);
```

The `public` keyword will make sure we generate the needed header so that we can call it from C. Calling from pure Python, we can work with this as if it was just any Python function. But the problem arises in C, where the return type is `PyObject *`, so you need to understand what you are actually returning and consult the Python API documentation to access the necessary data. I prefer keeping bindings between the languages simpler, as this is OK for void functions and will be easier. But if you want to return data, it can be frustrating. For example, from the above code snippet, if we know that we are returning an `int` type, we could use the following:

```
long returnValue = PyInt_AsLong (test (1, 0))
```

Notice the extra argument `__pyx_skip_dispatch`. As this is an implementation-specific argument, set this to 0 and your call should work the way you expect, taking the first parameter as the argument specified. The reason we use `long` is that any integer in Python is represented as `long`. You will need to refer to <http://docs.python.org/2/c-api/> for any other datatypes to get the data out of `PyObject`.

OpenMP support

OpenMP is a standard API in shared-memory parallel computing for languages; it's used in several open source projects such as Image Magick (<http://www.imagemagick.org/>) to try and speed up processing on large image manipulations. Cython has some support for this compiler extension. But you must be aware that you need to use compilers such as GCC or MSVC, which support OpenMP; Clang/LLVM has no OpenMP support yet. This isn't really a place to explain when and why to use OpenMP since it is really a vast subject, but you should check out the following link:

<http://docs.cython.org/src/userguide/parallelism.html>

Object initialization

You may remember from our garbage collection example earlier in the book that when defining a class in Cython, we have two initialization hooks for a class:

- `__cinit__`: Used for `cdef` initialization
- `__init__`: Used for normal Python initialization

Both of these have uses and sometimes you use both. The `__cinit__` hook is used before the Python `__init__` hook and is only called once! By convention, this is used to allocate memory for C structs wrapped as Python classes. But the `__init__` hook works just like any normal Python `__init__` hook and you can call this as much as you want to.

Compile time

At compile time, just like in C/C++, we have the C-preprocessor to make some decisions on what gets compiled, mostly from conditionals, defines, and a mixture of both. In Cython, we can replicate some of this behavior using `IF`, `ELIF`, `ELSE`, and `DEF`. This is demonstrated as an example in the following code line:

```
DEF myConstant = "hello cython"
```

We also have access to `os.uname` as predefined constants from the Cython compiler:

- `UNAME_SYSNAME`
- `UNAME_NODENAME`
- `UNAME_RELEASE`
- `UNAME_VERSION`
- `UNAME_MACHINE`

We can also run conditional expressions against these, such as:

```
IF UNAME_SYSNAME == "Windows":
    include "windows.pyx"
ELSE:
    include "unix.pyx"
```

You also have `ELIF` to use in conditional expressions. If you compare something like this against some of your headers in C programs, you will see how you can replicate basic C-preprocessor behavior in Cython. This gives you a quick idea of how you could replicate C-preprocessor usage in your headers.

Python 3

Porting to Python 3 can be painful, but reading around the subject shows us that people have had success porting their code to 3.x by simply compiling their module with Cython instead of actually porting their code! With Cython, you can specify the output to conform with the Python 3 API via:

```
10-4-5-52:~ redbrain$ cython -3 ...
```

This will make sure you are outputting Python 3 stuff instead of the default argument of `-2`, which generates for the 2.x standard.

Using PyPy

PyPy has become a popular alternative to the standard Python implementation. More importantly, it is now being used by many companies (small and large) in their production environments to boost performance and scalability. How does PyPy differ from normal CPython? While the latter is a traditional interpreter, the former is a full-fledged virtual machine. It maintains a just-in-time compiler backend for runtime optimization on most relevant architectures.

Getting Cythonized modules to run on PyPy is dependent on their `cpyext` emulation layer. This isn't quite complete and has many inconsistencies. But if you are brave and up to trying it out, it's going to get better and better with each release.

AutoPXD

When it comes to writing Cython modules, most of your work will comprise of getting your PXD declarations correct so that you can manipulate native code correctly. This would be a great addition to the Cython compiler, as you would be able to parse C/C++ headers and generate a PXD file. Part of my Google Summer of Code project was to use the Python plugin system as part of GCC to reuse GCC for parsing; the plugin could intercept the declarations and prototypes. The project isn't readily available at the moment as it suffered some problems, one being that the behavior of the plugin was dependent on the GCC version, and at that time, GCC 4.7 and 4.8 were undergoing a lot of internal changes as there was a transition of the language used, that is, from C to C++.

I intend to fully resurrect the project, polish it, and submit it back, as the GCC plugin system is starting to behave itself in different versions more sanely. There has been some effort in creating tools that are able to parse or compile headers to PXD declarations; you can check them out at <http://wiki.cython.org/AutoPxd>.

I will give you an idea of what's available and their statuses. But the main issue is developing, and more importantly, maintaining a fully compliant C and C++ parser. Also, utilizing a preprocessor is a main issue, which is why using GCC to do all that for us is so attractive.

Pyrex versus Cython

Cython is a derivative of Pyrex, though Pyrex has a much lower development pace compared to Cython; Pyrex also shows features similar to Cython. However, Pyrex is more primitive; Cython provides us with much more powerful typing and checks as well as optimizations and confidence with exception handling. It also provides more features such as `cpdef`. In the end, if you want to write native Python modules, you shouldn't really consider Pyrex anymore as it's mostly a dead project if you look at it, especially with Cython being its successor.

SWIG versus Cython

Overall, if you consider SWIG (<http://swig.org/>) as a way to write a native Python module, you could be fooled to think that Cython and SWIG are similar. SWIG is a very simple tool but is mainly used for writing wrappers for language bindings. For example, if you have some C code as follows:

```
int myFunction (int, const char *){ ... }
```

You would write the SWIG interface file as:

```
/* example.i */
%module example
%{
    extern int myFunction (int, const char *);
    ...
%}
```

Compile this with the following:

```
$ swig -python example.i
```

You can compile and link the module as you would do for a Cython output since this generates the necessary C code. This is fine if you want a basic module to simply call into C from Python. But Cython provides users with so much more. With SWIG, you will need to generally write wrappers for everything, even small interface functions going between each language.

Cython is much more developed and optimized, and it truly understands how to work with C types, memory management, and how to handle exceptions. With SWIG, you cannot manipulate data, you simply call into functions on the C side from Python. In Cython, we can call C from Python and vice versa. The type conversion is just so powerful; but not only this, we can also wrap C types into real Python classes to make C data feel Pythonic, which is very important when wanting to feel native to the language.

For example, remember the XML example from *Chapter 5, Advanced Cython*, where we were able to drop in the `import` replacement. This is possible because of Cython's type conversion, and the API is very Pythonic. Not only can we wrap C types into Pythonic objects, we also let Cython generate the boilerplate necessary for Python to do this without wrapping things into a class. What's more is that Cython produces very optimized code for the user.

Cython and NumPy

NumPy is a scientific library designed to provide functionality similar to or on par with MATLAB, which is a paid proprietary mathematics package. NumPy has a lot of popularity with Cython users since you can eek out more performance from your highly computational code by using C types. In Cython, you can import this library as follows:

```
import numpy as np
cimport numpy as np

np.import_array()
```

And you can access full Python APIs such as:

```
np.PyArray_ITER_NOTDONE
```

So, you can integrate with iterators at a very native area of the API. This allows NumPy users to get a lot of speed when working with native types via something like the following:

```
cdef double * val = (<double*>np.PyArray_MultiIter_DATA(it, 0))[0]
```

We can cast the data from the array to `double` and it's a `cdef` type in Cython to work with now. For more information and NumPy tutorials, visit the following link:

<http://wiki.cython.org/tutorials/numpy>

Numba versus Cython

Numba is another way of getting your Python code to become almost native to your host system by outputting the code to be run on LLVM seamlessly. Numba makes use of decorators such as:


```
@autojit
def myFunction (): ...
```

Numba also integrates with NumPy. On the whole, it sounds great. Unlike Cython, you only apply decorators to pure Python code and it does everything for you, but you may find that the optimizations will be fewer and not as powerful.

Numba does not integrate with C/C++ to the extent that Cython does. If you want to integrate, you need to use **Foreign Function Interfaces (FFI)** to wrap calls. You also need to define structs and work with C types in Python code in a very abstract sense to the point at which you don't really have much control as compared with Cython.

Numba is mostly comprised of decorators, such as `@locals`, from Cython. But in the end, all this creates just-in-time-compiled functions with a proper native function signature. Since you can specify the typing of function calls, this should provide more native speed when calling and returning data from functions. I would argue that the optimizations you will get as compared to Cython will be minimal as you might need a lot of abstractions to talk to native code, although calling in a lot of functions might be a faster technique.

Just for reference, LLVM is a low-level virtual machine; it's a compiler development infrastructure where projects can use it as a JIT compiler. The infrastructure can be extended to run things such as pure Java bytecode and even Python via Numba. It can be used for almost any purpose with a nice API for development. As apposed to GCC (an ahead-of-time compiler infrastructure), which implements a lot of static analysis ahead of time before code is run, LLVM allows code to change at runtime.

 For more information on Numba and LLVM, you can refer to the following links:

- <http://numba.pydata.org/>
- <http://llvm.org/>

Parakeet and Numba

Parakeet is another project that works alongside Numba, adding extremely specific optimizations to Python code that use lots of nested loops and parallelism. As with OpenMP, where it's really cool, Numba too requires using annotations on your code to do all this for the programmer. The downside is that you won't just magically optimize any Python code, the optimization that Parakeet does is on very specific sets of code.

GCCPy Python frontend to GCC

This is my pet project, which is a full-fledged ahead-of-time Python compiler that takes pure Python code on domain-specific sets and compiles it to an assembler, just like you would with C code. The project shows signs of working on basic sets of Python, in the last few months of 2013, as everything has been implemented from scratch (parser and runtime). What's interesting with this approach to Python is that your code is fully native; instead of the `.pyc` files, you have full-fledged `.so` shared objects on your Python modules. Therefore, deployment is simpler on embedded platforms; there is no need for virtual machines and hence there is lower memory usage. Consider virtual machines or interpreters that must handle so much more than compiling code to bytecode; if you compile ahead of time at runtime, you don't care about how the execution occurs because the runtime trusts the compiler to generate proper code.

Another benefit is that there have been a lot of speed ups to gain from by being fully native. This is still not ready for any real use. It has been inspired by PHC, which was a PHP-to-native compiler, but had some problems mostly due to doing even more from scratch by not re-using a compiler backend like GCC or LLVM. GCC also performs all of its optimizations on your Python code and backend optimizations such as instruction scheduling with `-mtune` core options with GCC.



For more information on this refer the following links:

- <http://gcc.gnu.org/wiki/PythonFrontEnd>
- <https://github.com/redbrain/gccpy>

Links and further reading

Some useful links for referencing are:

- <http://wiki.cython.org/FAQ>
- <http://wiki.cython.org/>
- <http://cython.org/>
- <http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/>
- <http://swig.org/>
- <http://www.numpy.org/>
- <http://wiki.cython.org/tutorials/numpy>
- <http://en.wikipedia.org/wiki/NumPy>
- <http://llvm.org/>
- <http://numba.pydata.org/>
- http://numba.pydata.org/numba-doc/0.9/interface_c.html
- <http://gcc.gnu.org/>

Summary

If you've read this far, you should now be familiar with Cython to such an extent that you can embed it with C bindings and even make some of your pure Python code more efficient. I've shown you how to apply Cython against an actual open source project, and even how to extend native software with a Twisted web server! As I kept saying through out the book, it makes C feel as though there are endless possibilities to control logic or that you can extend the system with the plethora of Python modules available. Thanks for reading.

Index

Symbols

`~Car` destructor 63
`__cinit__` hook 37, 84
`@cython.locals` annotation 77
`__dealloc__` hook 38
`-embed` option 49
`-enable-python` switch 32, 40
`__init__` hook 37, 84
`__str__` hook 38

A

annotations
used, for Cython typing 77, 78
args structure 45
atomic instructions 70
AutoPXD
about 86
URL 86

B

bool type 58, 66
BSD libc extension 44
build systems
integrating with 31-33
build systems, **integrating**
GNU/Autotools 32, 33
Python distutils 31

C

`callback_client_connect` function 27
Car class 66
Caveat 28
C++ bool type 66

C++ classes 62
C **const** 59
C++ **constructs**
about 61
allocation 63
bool type 66
classes 62
exceptions 64, 65
namespaces 61, 62
new keyword 63
overloading 66, 67
static class member attribute 68
templates 67
cdef class 37
cdef Cython functions 28
cdef function 22, 56, 77, 78
cdef keyword 22
cdef struct 62
C++ **exceptions**
about 64
handling 64, 65
CFLAGS option 42
cfunc annotation 78
C **functions**
calling in 68
C++ **functions**
calling in 68
classes 62
classobj foobar 53
`cmd_entry` definition 39
`cmd_entry` initialization hook 47
`cmd_entry_python` extern declaration 48
`cmd-kill-window.c` command 39
`cmd_table` command 38
C++ **namespaces**
about 61, 62

- using 69
- compile time** 84, 85
- ConfigParser standard libraries** 15
- configure -enable-python option** 41
- cpdef feature** 86
- cpdef function** 77
- cpdef keyword** 83, 84
- cpp class** 77
- cppclass name [T] syntax** 67
- cpyext emulation layer** 85
- C Python exceptions**
 - about 56, 57
 - URL 57
- C sockets**
 - creating, libevent used 26, 27
- C sockets, creating**
 - Cython callbacks 28
 - Cython PXD 28, 29
 - messaging engine 28
 - Python messaging engine 29, 30
- C types**
 - for loops 57
- C++ usage**
 - caveats 68
- C++ usage, caveats**
 - C functions, calling in 68
 - C++ functions, calling in 68
 - namespaces 69
 - Python distutils 69
- cygdb**
 - about 51
 - running 52-54
- Cython**
 - about 7
 - callbacks 28
 - compile time 84, 85
 - installing 8
 - pure Python code 35, 36
 - pure Python code, example 35
 - PXD 28, 29
 - typing, via annotations 77, 78
 - unit testing 75, 76
 - used, for Python code compiling 49
 - versus Numba 88
 - versus Pyrex 86
 - versus SWIG 86, 87
- Cython callbacks** 28

- Cython caveats**
 - * operator 55
 - about 54
 - bool type 58
 - C const 59
 - C Python exceptions 56, 57
 - C types for loops 57
 - multiple Cython files 59
 - struct, initializing 59
 - type checking 55
- Cython cdef**
 - about 15, 16
 - linking models 16, 17
 - public keyword 17, 18
 - Python, logging into 18-20
 - syntax 21
 - usage reference 21, 22, 24
- Cython cdef syntax**
 - enums 24
 - structs 22-24
- Cython installation**
 - about 8
 - code examples 9
 - emacs mode 8
- Cython keywords**
 - about 71
 - gil keyword 71
 - nogil keyword 71
- Cython PXD** 28, 29
- Cython support**
 - OpenMP 84

D

- data**
 - parsing 78-80

E

- emacs mode** 8
- exec hook** 40
- exec hook** 44
- extern void initythonfile (void) function** 17

F

- Fedora** 8
- Foreign Function Interfaces (FFI)** 88

for loops
on C types 57, 58
func function 52, 53
function pointers 25

G

garbage collector, Python 37, 38
GDB
about 51
cygdb, running 52-54
using on code 51
gdb commands 52
getopt usage 15
GIL
about 16, 69
atomic instructions 70
Cython keywords 71
messaging server 71-74
Read/write lock 70
Global Interpreter Lock. *See* GIL
globals command 53
GNU/Autotools 32, 33
GNU Project Debugger. *See* GDB

H

handleEvent event 30
Hello World 9, 10

I

Image Magick
URL 84
info () macro 18

L

libevent
about 26
used, for creating C sockets 26-30
libevent callbacks 74
libevent socket server 72
LIBS option 42
LinkedList template class 67
linking models
about 16
fully embedded Python 16

fully embedded Python, figure 16
Python shared object module 17
Python shared object module, figure 17

LLVM

URL 89

log_debug function 47

M

Mac 8
main() method 29
messaging engine 28
messaging server
using 71
web server, embedding 71-74
multiple Cython files 59
myFunc function 64

N

namespace keyword 61, 62
new keyword
about 63
allocation 63
Numba
URL 89
NumPy
about 87, 88
tutorials, URL 88

O

object
initializing 84
OpenMP support 84
overloading 66, 67
own module
calling, into C code 10-13
writing 10, 11

P

printf() method 18
printStruct function 23
public attributes 62
public keyword
about 17, 83
using 17, 18

- PYINCS variable** 32
- pyinit_server function** 28
- PYLIBS variable** 32
- PyObject** 7
- PyPy**
 - about 85
 - using 85
- Pyrex**
 - versus Cython 86
- Python**
 - about 7
 - bindings 36
 - embedding 42
 - garbage collector 37, 38
 - logging into 18-20
 - messaging engine 29, 30
 - URL 84
- Python 3**
 - AutoPXD 86
 - Cython 87, 88
 - NumPy 87, 88
 - porting to 85
 - PyPy, using 85
 - Pyrex versus Cython 86
 - SWIG versus Cython 86, 87
- Python code**
 - compiling 49
- Python ConfigParser**
 - about 20
 - working 20, 21
- Python distutils**
 - about 31
 - using 69
- Python.h header** 16
- Python IRC notifier**
 - about 75
 - URL 75
- Python messaging engine**
 - about 29
 - working 30
- Python modules**
 - calling into 60
 - size, adjusting 60
- Python threading**
 - about 69
 - atomic instructions 70
 - Cython keywords 71

- messaging server 71-74
- Read/write lock 70

R

- range function** 57
- Read/write lock** 70
- reference links** 90
- returns annotation** 78
- roster class** 29
- rosterEvent event** 30
- roster.handleEvent () method** 30
- RuntimeError exception** 65

S

- scalable asynchronous server**
 - about 26
 - diagram 26
- static class member attribute** 68
- static keyword** 68
- struct**
 - initializing 59
- struct args definition** 44
- struct cmd_entry**
 - cythonizing 43-46
- struct cmd_entry definition** 44
- subclassing**
 - preventing 77
- SWIG**
 - URL 86
 - versus Cython 86, 87

T

- templates** 67
- testStruct function** 23
- Tmux**
 - build system 40, 41
 - cmd_table command 38
 - extending 38-40
 - URL 38
- Tmux build system**
 - about 40
 - working 41
- Tmux command**
 - implementing 46, 47
- type checking**

about 55
example 55
Typedefs 25

U

Ubuntu/Debian 8

V

VA_ARGS function 47
void * pointer 55



Thank you for buying Learning Cython Programming

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

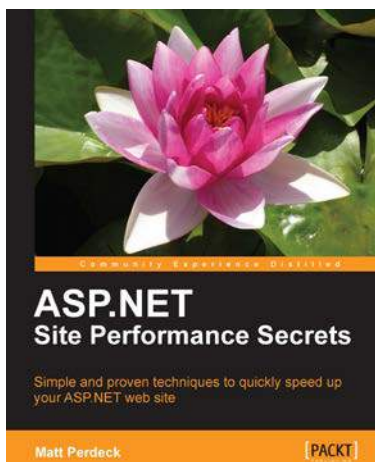


Expert Python Programming

ISBN: 978-1-84719-494-7 Paperback: 372 pages

Best practices for designing, coding, and distributing your Python software

1. Learn Python development best practices from an expert, with detailed coverage of naming and coding conventions
2. Apply object-oriented principles, design patterns, and advanced syntax tricks
3. Manage your code with distributed version control



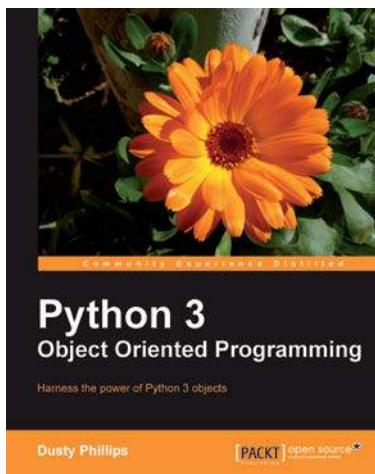
ASP.NET Site Performance Secrets

ISBN: 978-1-84969-068-3 Paperback: 456 pages

Simple and proven techniques to quickly speed up your ASP.NET website

1. Speed up your ASP.NET website by identifying performance bottlenecks that hold back your site's performance and fixing them
2. Tips and tricks for writing faster code and pinpointing those areas in the code that matter most, thus saving time and energy
3. Drastically reduce page load times

Please check www.PacktPub.com for information on our titles

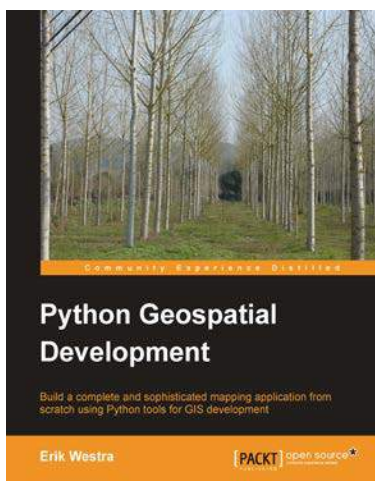


Python 3 Object Oriented Programming

ISBN: 978-1-84951-126-1 Paperback: 404 pages

Harness the power of Python 3 objects

1. Learn how to do Object Oriented Programming in Python using this step-by-step tutorial
2. Design public interfaces using abstraction, encapsulation, and information hiding
3. Turn your designs into working software by studying the Python syntax
4. Raise, handle, define, and manipulate exceptions using special error objects



Python Geospatial Development

ISBN: 978-1-84951-154-4 Paperback: 508 pages

Build a complete and sophisticated mapping application from scratch using Python tools for GIS development

1. Build applications for GIS development using Python
2. Analyze and visualize Geospatial data
3. Comprehensive coverage of key GIS concepts
4. Recommended best practices for storing spatial data in a database

Please check www.PacktPub.com for information on our titles