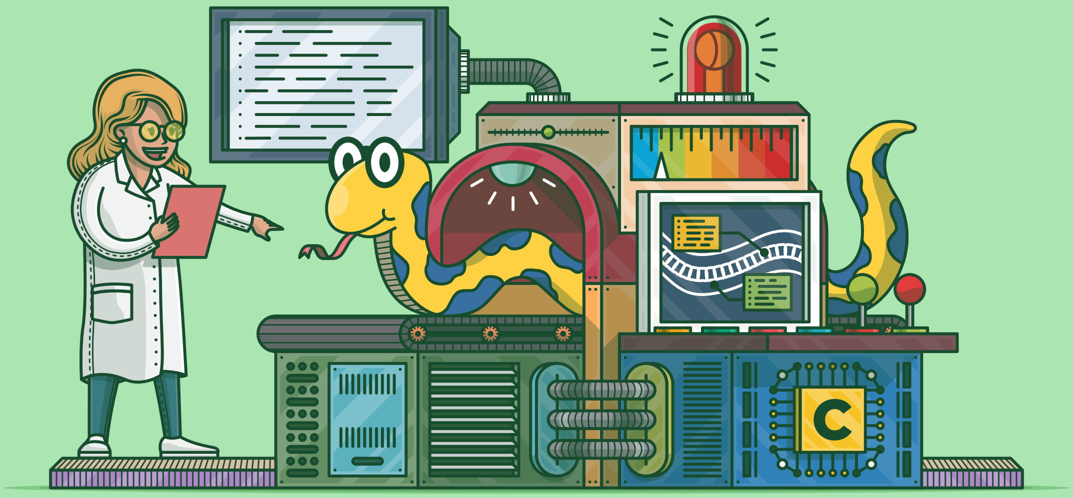
Real Python

CPYTHON INTERNALS



YOUR GUIDE TO THE PYTHON 3 INTERPRETER

FIRST EDITION

BY ANTHONY SHAW AND THE REALPYTHON.COM TUTORIAL TEAM

CPython Internals: Your Guide to the Python 3 Interpreter

Anthony Shaw

CPython Internals: Your Guide to the Python 3 Interpreter

Anthony Shaw

ISBN: 9781775093343 (paperback)

ISBN: 9781775093350 (electronic)

Cover design by Aldren Santos

Additional editing and proofreading by Jacob Schmitt

What Readers Say About *CPython Internals: Your Guide to the Python 3 Interpreter*

"It's the book that I wish existed years ago when I started my Python journey. After reading this book your skills will grow and you will be able solve even more complex problems that can improve our world."

— **Carol Willing**, CPython core developer and member of the CPython Steering Council

"The 'Parallelism and Concurrency' chapter is one of my favorites. I had been looking to get an in depth understanding around this topic and I found your book extremely helpful.

Of course, after going over that chapter I couldn't resist the rest. I am eagerly looking forward to have my own printed copy once it's out!

I had gone through your 'Guide to the CPython Source Code' article previously, which got me interested in finding out more about the internals.

There are a ton of books on Python which teach the language, but I haven't really come across anything that would go about explaining the internals to those curious minded.

And while I teach Python to my daughter currently, I have this book added in her must-read list. She's currently studying information systems at Georgia State University."

— **Milan Patel**, vice president at (a major investment bank)

"What impresses me the most about Anthony's book is how it puts all the steps for making changes to the CPython code base in an easy-to-follow sequence. It really feels like a 'missing manual' of sorts.

Diving into the C underpinnings of Python was a lot of fun and it cleared up some longstanding questions marks for me. I found the chapter about CPython's memory allocator especially enlightening.

CPython Internals is a great (and unique) resource for anybody looking to take their knowledge of Python to a deeper level."

— **Dan Bader**, author of *Python Tricks* and editor in chief at *Real Python*

"This book helped me to better understand how lexing and parsing works in Python. It's my recommended source if you want to understand it."

— **Florian Dahlitz**, Pythonista

"A comprehensive walkthrough of the Python internals, a topic which surprisingly has almost no good resource, in an easy-to-understand manner for both beginners as well as advanced Python users."

— **Abhishek Sharma**, data scientist

About the Author

Anthony Shaw is an avid Pythonista and Fellow of the Python Software Foundation.

Anthony has been programming since the age of 12 and found a love for Python while trapped inside a hotel in Seattle, Washington, 15 years later. After ditching the other languages he'd learned, Anthony has been researching, writing about, and creating courses for Python ever since.

Anthony also contributes to small and large Open Source projects, including CPython, as well as being a member of the Apache Software Foundation.

Anthony's passion lies in understanding complex systems, then simplifying them, and teaching them to people.

About the Review Team

Jim Anderson has been programming for a long time in a variety of languages. He has worked on embedded systems, built distributed build systems, done off-shore vendor management, and sat in many, many meetings.

Joanna Jablonski is the executive editor of *Real Python*. She likes natural languages just as much as she likes programming languages. Her love for puzzles, patterns, and pesky little details led her to follow a career in translation. It was only a matter of time before she would fall in love with a new language: Python! She joined *Real Python* in 2018 and has been helping Pythonistas level up ever since.

Contents

Contents 7

Foreword 12

Introduction 14

How to Use This Book 15
Bonus Material and Learning Resources 17

Getting the CPython Source Code 21

What's in the Source Code? 22

Setting Up Your Development Environment 24

IDE or Editor? 24
Setting Up Visual Studio 26
Setting Up Visual Studio Code 28
Setting Up JetBrains CLion 33
Setting up Vim 37
Conclusion 41

Compiling CPython **43**
Compiling CPython on macOS 44
Compiling CPython on Linux 46
Installing a Custom Version 48
A Quick Primer on Make 48
CPython's Make Targets 50
Compiling CPython on Windows 53
Profile-Guided Optimization 59
Conclusion 61

The Python Language and Grammar **62**
Why CPython Is Written in C and Not Python 63
The Python Language Specification 65
The Parser Generator 70
Regenerating Grammar 70
Conclusion 75

Configuration and Input **77**
Configuration State 80
Build Configuration 84
Building a Module From Input 85
Conclusion 90

Lexing and Parsing With Syntax Trees **92**
Concrete Syntax Tree Generation 93
The CPython Parser-Tokenizer 97
Abstract Syntax Trees 102
Important Terms to Remember 112
Example: Adding an Almost-Equal Comparison Operator . 112
Conclusion 118

The Compiler **119**
Related Source Files 120
Important Terms 121
Instantiating a Compiler 122
Future Flags and Compiler Flags 123
Symbol Tables 124
Core Compilation Process 131
Assembly 138
Creating a Code Object 142
Using Instaviz to Show a Code Object 143
Example: Implementing the Almost-Equal Operator 145
Conclusion 151

The Evaluation Loop **152**
Related Source Files 153
Important Terms 153
Constructing Thread State 154
Constructing Frame Objects 155
Frame Execution 163
The Value Stack 166
Example: Adding an Item to a List 172
Conclusion 176

Memory Management **178**
Memory Allocation in C 179
Design of the Python Memory Management System 182
The CPython Memory Allocator 184
The Object and PyMem Memory Allocation Domains . . . 194
The Raw Memory Allocation Domain 197
Custom Domain Allocators 198
Custom Memory Allocation Sanitizers 199
The PyArena Memory Arena 202
Reference Counting 203
Garbage Collection 210
Conclusion 220

Parallelism and Concurrency 222
- Models of Parallelism and Concurrency 224
- The Structure of a Process 224
- Multiprocess Parallelism 227
- Multithreading 251
- Asynchronous Programming 266
- Generators 266
- Coroutines 273
- Asynchronous Generators 279
- Subinterpreters 280
- Conclusion 284

Objects and Types 286
- Examples in This Chapter 287
- Built-in Types 288
- Object and Variable Object Types 289
- The `type` Type 290
- The `bool` and `long` Types 294
- The Unicode String Type 299
- The Dictionary Type 310
- Conclusion 316

The Standard Library 317
- Python Modules 317
- Python and C Modules 319

The Test Suite 323
- Running the Test Suite on Windows 323
- Running the Test Suite on Linux or macOS 324
- Test Flags 325
- Running Specific Tests 325
- Testing Modules 327
- Test Utilities 328
- Conclusion 329

Debugging **330**
Using the Crash Handler 331
Compiling Debug Support 331
Using LLDB for macOS 332
Using GDB 336
Using Visual Studio Debugger 339
Using CLion Debugger 341
Conclusion 346

Benchmarking, Profiling, and Tracing **347**
Using `timeit` for Microbenchmarks 348
Using the Python Benchmark Suite for Runtime Benchmarks 350
Profiling Python Code with `cProfile` 356
Profiling C Code with DTrace 359
Conclusion 364

Next Steps **365**
Writing C Extensions for CPython 365
Improving Your Python Applications 366
Contributing to the CPython Project 367
Keep Learning 370

Appendix: Introduction to C for Python Programmers **372**
The C Preprocessor 372
Basic C Syntax 376
Conclusion 383

Foreword

> A programming language created by a community fosters happiness in its users around the world.
>
> — Guido van Rossum, “King’s Day Speech”

I love building tools that help us learn, empower us to create, and move us to share knowledge and ideas with others. I feel humbled, thankful, and proud when I hear how these tools and Python are helping you to solve real-world problems, like climate change or Alzheimer’s.

Through my four-decade love of programming and problem solving, I have spent time learning, writing a lot of code, and sharing my ideas with others. I’ve seen profound changes in technology as the world has progressed from mainframes to cell phone service to the wide-ranging wonders of the Web and cloud computing. All these technologies, including Python, have one thing in common.

At one moment, these successful innovations were nothing more than an idea. The creators, like Guido, had to take risks and leaps of faith to move forward. Dedication, learning through trial and error, and working together through many failures built a solid foundation for success and growth.

CPython Internals will take you on a journey to explore the wildly successful programming language **Python**. The book serves as a guide to how CPython works under the hood. It will give you a glimpse of how the core developers crafted the language.

Python's strengths include its readability and the welcoming community dedicated to education. Anthony embraces these strengths when explaining CPython, encouraging you to read the source and sharing the building blocks of the language with you.

Why do I want to share Anthony's *CPython Internals* with you? It's the book that I wish existed years ago when I started my Python journey. More importantly, I believe we, as members of the Python community, have a unique opportunity to put our expertise to work to help solve the complex real-world problems facing us.

I'm confident that after reading this book, your skills will grow, and you will be able solve even more complex problems and improve our world.

It's my hope that Anthony motivates you to learn more about Python, inspires you to build innovative things, and gives you confidence to share your creations with the world.

> Now is better than never.
>
> — Tim Peters, *The Zen of Python*

Let's follow Tim's wisdom and get started now.

Warmly,

— **Carol Willing**, CPython core developer and member of the CPython Steering Council

Introduction

Are there certain parts of Python that just seem like magic, like how finding an item is so much faster with dictionaries than looping over a list? How does a generator remember the state of variables each time it yields a value? Why don't you ever have to allocate memory like you do with other languages?

The answer is that CPython, the most popular Python runtime, is written in human-readable C and Python code.

CPython abstracts the complexities of the underlying C platform and your operating system. It makes threading straightforward and cross-platform. It takes the pain of memory management in C and makes it simple.

CPython gives the developer writing Python code the platform to write scalable and performant applications. At some stage in your progression as a Python developer, you'll need to understand how CPython works. These abstractions aren't perfect, and they're leaky.

Once you understand how CPython works, you can fully leverage its power and optimize your applications. This book will explain the concepts, ideas, and technicalities of CPython.

In this book, you'll cover the major concepts behind the internals of CPython and learn how to:

- Read and navigate the source code
- Compile CPython from source code

- Make changes to the Python syntax and compile them into your version of CPython
- Navigate and comprehend the inner workings of features like lists, dictionaries, and generators
- Master CPython's memory management capabilities
- Scale your Python code with parallelism and concurrency
- Modify the core types with new functionality
- Run the test suite
- Profile and benchmark the performance of your Python code and runtime
- Debug C and Python code like a professional
- Modify or upgrade components of the CPython library to contribute them to future versions

Take your time with each chapter and try out the demos and interactive elements. You'll feel a sense of achievement as you grasp the core concepts that will make you a better Python programmer.

How to Use This Book

This book is all about learning by doing, so be sure to set up your IDE early on by reading the instructions, downloading the code, and writing the examples.

For the best results, we recommend that you avoid copying and pasting the code examples. The examples in this book took many iterations to get right, and they may also contain bugs.

Making mistakes and learning how to fix them is part of the learning process. You might discover better ways to implement the examples, try changing them, and see what effect it has.

With enough practice, you'll master this material—and have fun along the way!

How skilled in Python do I need to be to use this book?

This book is aimed at intermediate to advanced Python developers. Every effort has been taken to show code examples, but some intermediate Python techniques will be used throughout.

Do I need to know C to use this book?

You don't need to be proficient in C to use this book. If you're new to C, then check out the appendix, "Introduction to C for Python Programmers," for a quick introduction.

How long will it take to finish this book?

We don't recommend rushing through this book. Try reading one chapter at a time, trying the examples after each chapter and exploring the code simultaneously. Once you've finished the book, it will make a great reference guide for you to come back to in time.

Won't the content in this book be out of date really quickly?

Python has been around for more than thirty years. Some parts of the CPython code haven't been touched since they were originally written. Many of the principles in this book have been the same for ten or more years.

In fact, while writing this book, we discovered many lines of code that were written by Guido van Rossum (the author of Python) and left untouched since version 1.

Some of the concepts in this book are brand-new. Some are even experimental. While writing this book, we came across issues in the source code and bugs in CPython that were later fixed or improved. That's part of the wonder of CPython as a flourishing open source project.

The skills you'll learn in this book will help you read and understand current and future versions of CPython. Change is constant, and expertise is something you can develop along the way.

Bonus Material and Learning Resources

This book comes with a number of free bonus resources that you can access at realpython.com/cpython-internals/resources/. On this web page you can also find an errata list with corrections maintained by the *Real Python* team.

Code Samples

The examples and sample configurations throughout this book will be marked with a header denoting them as part of the `cpython-book-samples` folder:

`cpython-book-samples ▸ 01 ▸ example.py`

```python
import this
```

You can download the code samples at realpython.com/cpython-internals/resources/.

Code Licenses

The example Python scripts associated with this book are licensed under a Creative Commons Public Domain (CC0) License. This means you're welcome to use any portion of the code for any purpose in your own programs.

CPython is licensed under the Python Software Foundation 2.0 license. Snippets and samples of CPython source code used in this book are done so under the terms of the PSF 2.0 license.

Note

The code in this book has been tested with Python 3.9 on Windows 10, macOS 10.15, and Linux.

Formatting Conventions

Code blocks are used to present example code:

```
# This is Python code:
print("Hello, World!")
```

Operating system–agnostic commands follow the Unix-style format:

```
$ # This is a terminal command:
$ python hello-world.py
```

(The `$` is not part of the command.)

Windows-specific commands have the Windows command-line format:

```
> python hello-world.py
```

(The `>` is not part of the command.)

Command-line syntax follows this format:

- `Unbracketed text` must be typed as it is shown.
- `<Text inside angle brackets>` indicates a variable for which you must supply a value. For example, you would replace `<filename>` with the name of a specific file.
- `[Text inside square brackets]` indicates an optional argument that you may supply.

Bold text denotes a new or important term.

Notes and alert boxes appear as follows:

Note

This is a note filled in with placeholder text. The quick brown fox jumps over the lazy dog. The quick brown Python slithers over the lazy hog.

Important

This is an alert also filled in with placeholder text. The quick brown fox jumps over the lazy dog. The quick brown Python slithers over the lazy hog.

Any references to a file within the CPython source code will be shown like this:

`path ▸ to ▸ file.py`

Shortcuts or menu commands will be given in sequence, like this:

File ❯ Other ❯ Option

Keyboard commands and shortcuts will be given for both macOS and Windows:

Ctrl + Space

Feedback and Errata

We welcome ideas, suggestions, feedback, and the occasional rant. Did you find a topic confusing? Did you find an error in the text or code? Did we leave out a topic you would love to know more about?

We're always looking to improve our teaching materials. Whatever the reason, please send in your feedback at the link below:

realpython.com/cpython-internals/feedback

About Real Python

At *Real Python*, you'll learn real-world programming skills from a community of professional Pythonistas from all around the world.

The realpython.com website launched in 2012 and currently helps more than three million Python developers each month with books, programming tutorials, and other in-depth learning resources.

Here's where you can find *Real Python* on the Web:

- realpython.com
- @realpython on Twitter
- The *Real Python Newsletter*
- The *Real Python Podcast*

Getting the CPython Source Code

When you type `python` at the console or install a Python distribution from Python.org, you're running **CPython**. CPython is one of many Python implementations maintained and written by different teams of developers. Some alternatives you may have heard of are PyPy, Cython, and Jython.

The unique thing about CPython is that it contains both a runtime and the shared language specification that all other Python implementations use. CPython is the official, or reference, implementation of Python.

The **Python language specification** is the document that describes the Python language. For example, it says that `assert` is a reserved keyword and that `[]` is used for indexing, slicing, and creating empty lists.

Think about the features you expect from the Python distribution:

- When you type `python` without a file or module, it gives an interactive prompt (REPL).
- You can import built-in modules like `json`, `csv`, and `collections` from the standard library.
- You can install packages from the Internet using `pip`.
- You can test your applications using the built-in `unittest` library.

These are all part of the CPython distribution. It includes a lot more than just a compiler.

In this book, you’ll explore the different parts of the CPython distribution:

- The language specification
- The compiler
- The standard library modules
- The core types
- The test suite

What’s in the Source Code?

The CPython source distribution comes with a whole range of tools, libraries, and components that you’ll explore in this book.

Note

This book targets version 3.9 of the CPython source code.

To download a copy of the CPython source code, you can use `git` to pull the latest version:

```
$ git clone --branch 3.9 https://github.com/python/cpython
$ cd cpython
```

The examples in this book are based on Python version 3.9.

Important

Switching to the 3.9 branch is an important step. The master branch changes on an hourly basis. Many of the examples and exercises in this book are unlikely to work on master.

Note

If you don't have Git available, then you can install it from git-scm.com. Alternatively, you can download a ZIP file of the CPython source directly from the GitHub website.

If you download the source as a ZIP file, then it won't contain any history, tags, or branches.

Inside the newly downloaded `cpython` directory, you'll find the following subdirectories:

cpython/

Doc	Source for the documentation
Grammar	The computer-readable language definition
Include	The C header files
Lib	Standard library modules written in Python
Mac	macOS support files
Misc	Miscellaneous files
Modules	Standard library modules written in C
Objects	Core types and the object model
Parser	The Python parser source code
PC	Windows build support files for older versions of Windows
PCBuild	Windows build support files
Programs	Source code for the `python` executable and other binaries
Python	The CPython interpreter source code
Tools	Standalone tools useful for building or extending CPython
m4	Custom scripts to automate configuration of the makefile

Next, you'll set up your development environment.

Setting Up Your Development Environment

Throughout this book, you'll be working with both C and Python code. It's essential that you have your development environment configured to support both languages.

The CPython source code is about 65 percent Python (of which the tests are a significant part) and 24 percent C. The remainder is a mix of other languages.

IDE or Editor?

If you haven't yet decided which development environment to use, then there's one decision to make first: whether to use an integrated development environment (IDE) or a code editor.

- An **IDE** targets a specific language and toolchain. Most IDEs have integrated testing, syntax checking, version control, and compilation.
- A **code editor** enables you to edit code files, regardless of language. Most code editors are simple text editors with syntax highlighting.

Because of their full-featured nature, IDEs often consume more hardware resources. So if you have limited RAM (less than 8 GB), then a code editor is recommended.

IDEs also take longer to start up. If you want to edit a file quickly, then a code editor is a better choice.

There are hundreds of editors and IDEs available for free or at a cost. Here are some commonly used IDEs and editors suitable for CPython development:

Application	Style	Supports
Microsoft Visual Studio Code	Editor	Windows, macOS, and Linux
Atom	Editor	Windows, macOS, and Linux
Sublime Text	Editor	Windows, macOS, and Linux
Vim	Editor	Windows, macOS, and Linux
Emacs	Editor	Windows, macOS, and Linux
Microsoft Visual Studio	IDE (C, Python, and others)	Windows
PyCharm by JetBrains	IDE (Python and others)	Windows, macOS, and Linux
CLion by JetBrains	IDE (C and others)	Windows, macOS, and Linux

A version of Microsoft Visual Studio is also available for Mac, but it doesn't support Python Tools for Visual Studio or C compilation.

In the sections below, you'll explore the setup steps for the following editors and IDEs:

- Microsoft Visual Studio
- Microsoft Visual Studio Code
- JetBrains CLion
- Vim

Skip ahead to the section for your chosen application, or read all of them if you want to compare.

Setting Up Visual Studio

The newest version of Visual Studio, Visual Studio 2019, has built-in support for Python and the C source code on Windows. I recommend using it for the examples and exercises in this book. If you already have Visual Studio 2017 installed, then that would also work.

Note

None of the paid features of Visual Studio are required for compiling CPython or completing this book. You can use the free Community edition.

However, the profile-guided optimization build profile requires the Professional edition or higher.

Visual Studio is available for free from Microsoft's Visual Studio website.

Once you've downloaded the Visual Studio installer, you'll be asked to select which components you want to install. You'll need the following components for this book:

- The **Python development** workload
- The optional **Python native development tools**
- Python 3 64-bit (3.7.2)

You can deselect Python 3 64-bit (3.7.2) if you already have Python 3.7 installed. You can also deselect any other optional features if you want to conserve disk space.

The installer will then download and install all the required components. The installation can take up to an hour, so you may want to read on and come back to this section when it finishes.

Once the installation is complete, click `Launch` to start Visual Studio. You'll be prompted to sign in. If you have a Microsoft account, you can either log in or skip that step.

Next, you'll be prompted to open a project. You can clone CPython's Git repository directly from Visual Studio by choosing the `Clone or check out code` option.

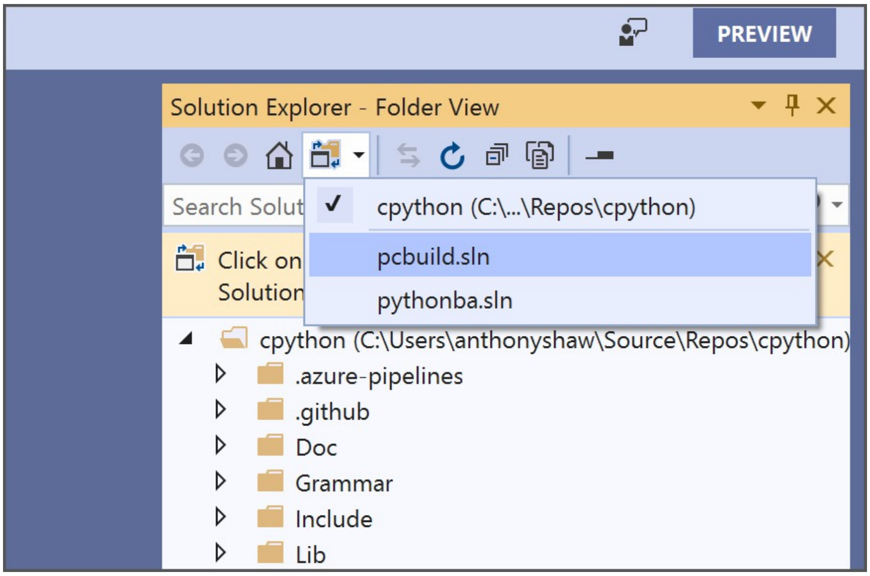
For the repository location, enter `https://github.com/python/cpython`, choose your local path, and select `Clone`.

Visual Studio will then download a copy of CPython from GitHub using the version of Git bundled with Visual Studio. This step also saves you the hassle of having to install Git on Windows. The download may take up to ten minutes.

Important

Visual Studio will automatically checkout the master branch. Before compiling, make sure you change to the 3.9 branch from within the Team Explorer window. Switching to the 3.9 branch is an important step. The master branch changes on an hourly basis. Many of the examples and exercises in this book are unlikely to work on master.

Once the project has downloaded, you need to point Visual Studio to the `PCBuild`▸`pcbuild.sln` solution file by clicking `Solutions and Projects` › `pcbuild.sln`:


PREVIEW
Solution Explorer - Folder View
Search Solut
Click on
Solution
✓ cpython (C:\...\Repos\cpython)
pcbuild.sln
pythonba.sln
cpython (C:\Users\anthonyshaw\Source\Repos\cpython)
.azure-pipelines
.github
Doc
Grammar
Include
Lib

Now that you have Visual Studio configured and the source code downloaded, you can compile CPython on Windows by following the steps in the next chapter.

Setting Up Visual Studio Code

Microsoft Visual Studio Code is an extensible code editor with an online marketplace of plugins.

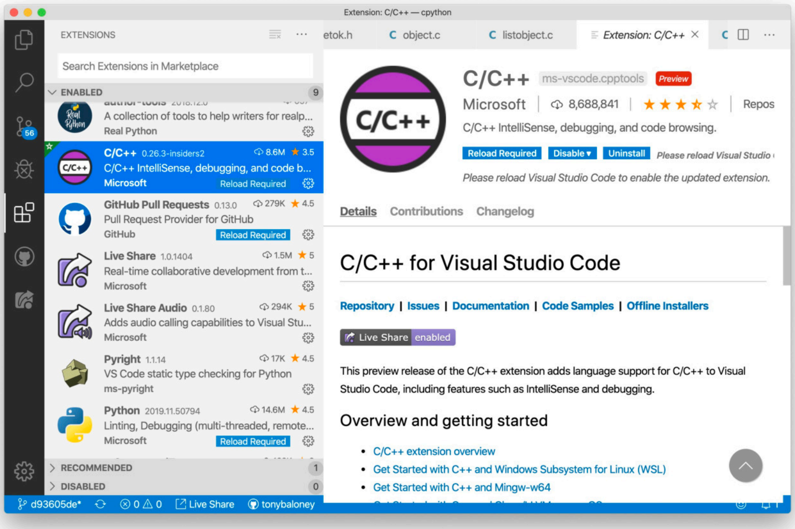
It makes an excellent choice for working with CPython as it supports both C and Python with an integrated Git interface.

Installing

Visual Studio Code, sometimes known as VS Code, is available with a simple installer at code.visualstudio.com.

Out of the box, VS Code has the necessary code editing capabilities, but it becomes more powerful once you install extensions.

You can access the Extensions panel by selecting View ⟩ Extensions from the top menu:


Extension: C/C++ — cpython
EXTENSIONS
Search Extensions in Marketplace
ENABLED 9
A collection of tools to help writers for realp...
Real Python
C/C++ 0.26.3-insiders2 8.6M 3.5
C/C++ IntelliSense, debugging, and code b...
Microsoft Reload Required
GitHub Pull Requests 0.13.0 279K 4.5
Pull Request Provider for GitHub
GitHub Reload Required
Live Share 1.0.1404 1.5M 5
Real-time collaborative development from t...
Microsoft
Live Share Audio 0.1.80 294K 5
Adds audio calling capabilities to Visual Stu...
Microsoft
Pyright 1.1.14 17K 4.5
VS Code static type checking for Python
ms-pyright
Python 2019.11.50794 14.6M 4.5
Linting, Debugging (multi-threaded, remote...
Microsoft Reload Required
RECOMMENDED 1
DISABLED 0
d93605de* ⊗ 0 ⚠ 0 Live Share tonybaloney
etok.h
object.c
listobject.c
Extension: C/C++
C/C++ ms-vscode.cpptools Preview
Microsoft | 8,688,841 | ★★★★☆ | Repos
C/C++ IntelliSense, debugging, and code browsing.
Reload Required Disable ▾ Uninstall
Details Contributions Changelog
Repository | Issues | Documentation | Code Samples | Offline Installers
Live Share enabled
C/C++ extension overview
Get Started with C++ and Windows Subsystem for Linux (WSL)
Get Started with C++ and Mingw-w64

Please reload Visual Studio Code to enable the updated extension.

C/C++ for Visual Studio Code

This preview release of the C/C++ extension adds language support for C/C++ to Visual Studio Code, including features such as IntelliSense and debugging.

Overview and getting started

Inside the Extensions panel, you can search for extensions by name or by their unique identifier, such as `ms-vscode.cpptools`. In some cases there are many plugins with similar names, so use the unique identifier to be sure you're installing the right one.

Recommended Extensions for This Book

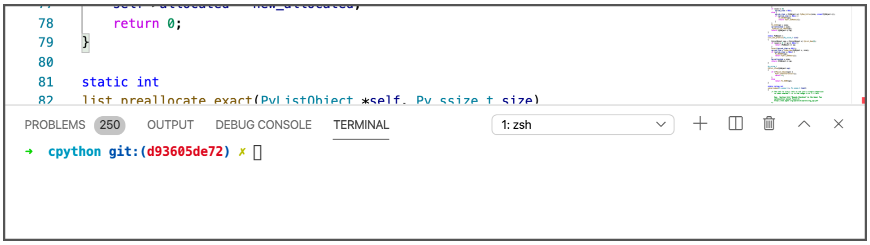
There are several useful extensions for working with CPython:

- **C/C++ (`ms-vscode.cpptools`)** provides support for C/C++, including IntelliSense, debugging, and code highlighting.
- **Python (`ms-python.python`)** provides rich Python support for editing, debugging, and reading Python code.
- **reStructuredText (`lextudio.restructuredtext`)** provides rich support for reStructuredText, the format used in the CPython documentation.

- **Task Explorer (`spmeesseman.vscode-taskexplorer`)** adds a Task Explorer panel inside the Explorer tab, making it easier to launch `make` tasks.

After you install these extensions, you'll need to reload the editor.

Many of the tasks in this book require a command line. You can add an integrated terminal into VS Code by selecting Terminal > New Terminal. A terminal will appear below the code editor:



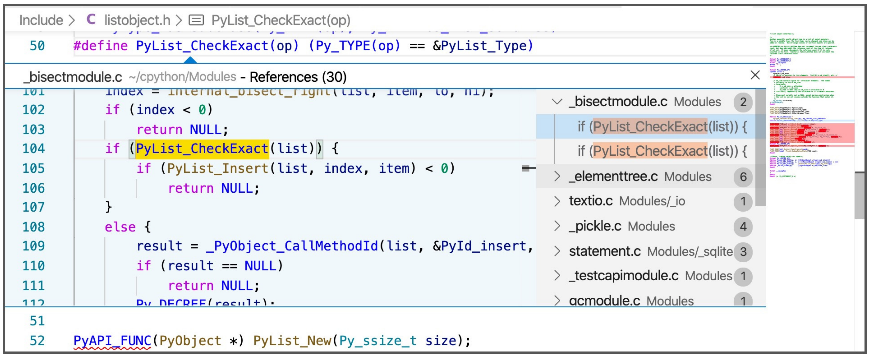
```
    return 0;
}

static int
list_preallocate_exact(PyListObject *self, Py_ssize_t size)
```

```
➜ cpython git:(d93605de72) ✗
```

Using Advanced Code Navigation and Expansion

With the plugins installed, you can perform some advanced code navigation.

For example, if you right-click a function call in a C file and select Go to References, then VS Code will find other references to that function in the codebase:



Include > listobject.h > PyList_CheckExact(op)

```
#define PyList_CheckExact(op) (Py_TYPE(op) == &PyList_Type)
```

_bisectmodule.c ~/cpython/Modules - References (30)

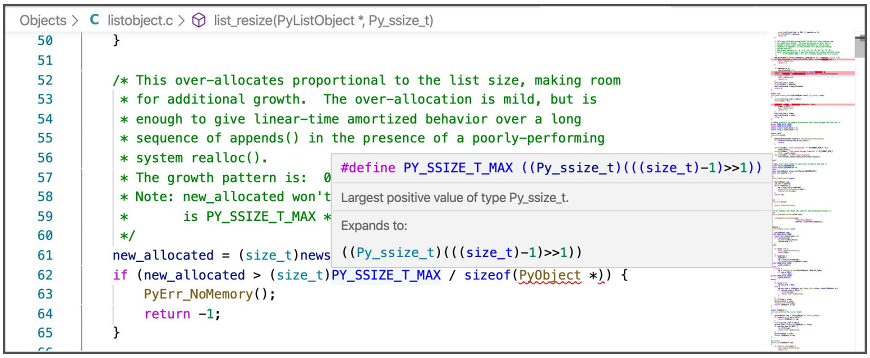
```
    index = internal_bisect_right(list, item, lo, hi);
    if (index < 0)
        return NULL;
    if (PyList_CheckExact(list)) {
        if (PyList_Insert(list, index, item) < 0)
            return NULL;
    }
    else {
        result = _PyObject_CallMethodId(list, &PyId_insert,
        if (result == NULL)
            return NULL;
        Py_DECREF(result);
```

_bisectmodule.c Modules	2
if (PyList_CheckExact(list)) {	
if (PyList_CheckExact(list)) {	
_elementtree.c Modules	6
textio.c Modules/_io	1
_pickle.c Modules	4
statement.c Modules/_sqlite	3
_testcapimodule.c Modules	1
gcmodule.c Modules	1

```
PyAPI_FUNC(PyObject *) PyList_New(Py_ssize_t size);
```

Go to References is very useful for discovering the proper calling form for a function.

If you click on or hover over a C macro, then the editor will expand that macro to the compiled code:



Objects > C listobject.c > list_resize(PyListObject *, Py_ssize_t)

```
    }

    /* This over-allocates proportional to the list size, making room
     * for additional growth.  The over-allocation is mild, but is
     * enough to give linear-time amortized behavior over a long
     * sequence of appends() in the presence of a poorly-performing
     * system realloc().
     * The growth pattern is:  0
     * Note: new_allocated won't
     *       is PY_SSIZE_T_MAX *
     */
    new_allocated = (size_t)news
    if (new_allocated > (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)) {
        PyErr_NoMemory();
        return -1;
    }
```

```
#define PY_SSIZE_T_MAX ((Py_ssize_t)(((size_t)-1)>>1))
```

Largest positive value of type Py_ssize_t.

Expands to:

```
((Py_ssize_t)(((size_t)-1)>>1))
```

To jump to the definition of a function, hover over any call to it and press Cmd + Click on macOS or Ctrl + Click on Linux and Windows.

Configuring the Task and Launch Files

VS Code uses a `.vscode` folder in the workspace directory. If this folder doesn't exist, create it now. Inside this folder, you can create the following files:

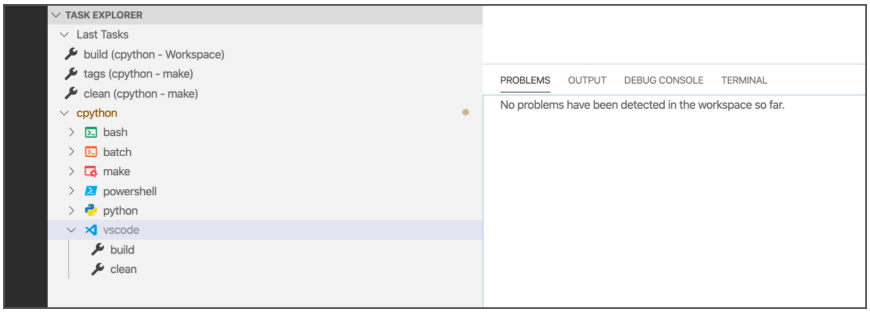
- **`tasks.json`** for shortcuts to commands that execute your project
- **`launch.json`** to configure the debugger (see the chapter "Debugging")
- Other plugin-specific files

Create a `tasks.json` file inside the `.vscode` directory if one doesn't already exist. This `tasks.json` file will get you started:

`cpython-book-samples▸11▸tasks.json`

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "type": "shell",
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "windows": {
                "command": "PCBuild/build.bat",
                "args": ["-p", "x64", "-c", "Debug"]
            },
            "linux": {
                "command": "make -j2 -s"
            },
            "osx": {
                "command": "make -j2 -s"
            }
        }
    ]
}
```

Using the Task Explorer plugin, you'll see a list of your configured tasks inside the `vscode` group:



No problems have been detected in the workspace so far.

In the next chapter, you'll learn more about the build process for compiling CPython.

Setting Up JetBrains CLion

JetBrains makes an IDE for Python called PyCharm as well as an IDE for C/C++ development called CLion.

CPython has both C and Python code. You can't install C/C++ support into PyCharm, but CLion comes bundled with Python support.

Important

Makefile support is available only in CLion versions 2020.2 and above.

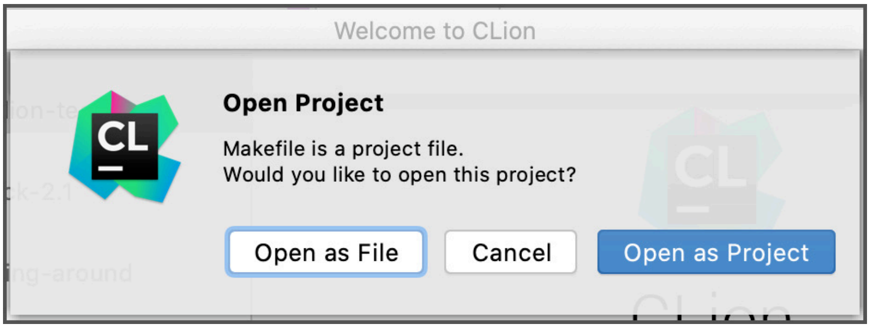
Important

This step requires that you have both generated a makefile by running `configure` and compiled CPython.

Please read the chapter "Compiling CPython" for your operating system and then return to this chapter.

After compiling CPython for the first time, you'll have a makefile in the root of the source directory.

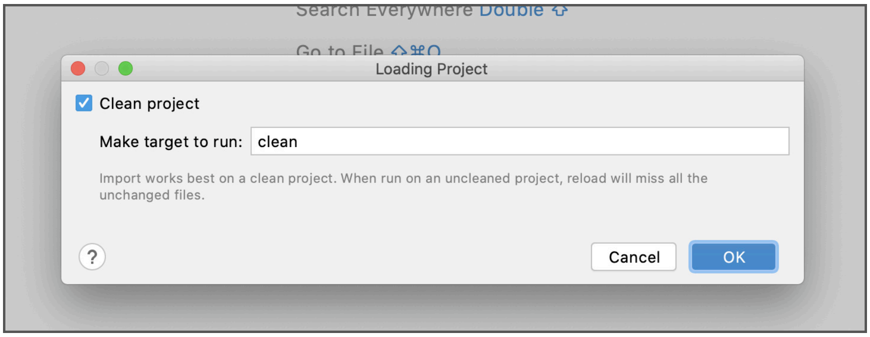
Open CLion and choose Open or Import from the welcome screen. Navigate to the source directory, select the makefile, and press Open:



Makefile is a project file.
Would you like to open this project?

CLion will ask whether you want to open the directory or import the makefile as a new project. Select Open as Project to import as a project.

CLion will ask which `make` target to run before importing. Leave the default option, `clean`, and continue:



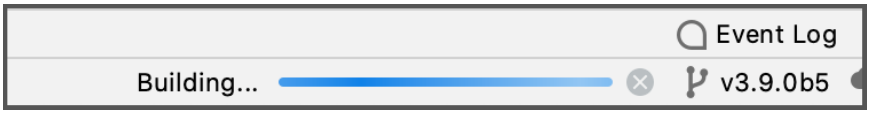
Clean project [✓]

Make target to run: clean

Import works best on a clean project. When run on an uncleaned project, reload will miss all the unchanged files.

Next, check that you can build the CPython executable from CLion. From the top menu, select Build › Build Project.

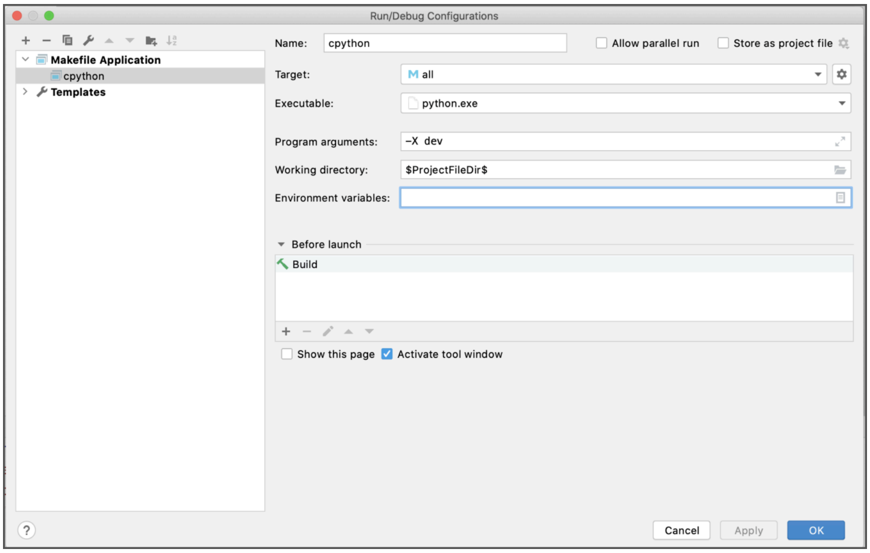
In the status bar, you should see a progress indicator for the project build:



Once this task is complete, you can target the compiled binary as a run/debug configuration.

Select Run > Edit Configurations to open the Run/Debug Configurations window. Inside this window, select + > Makefile Application and complete the following steps:

1. Set the `Name` to `cpython`.
2. Leave the build target as `all`.
3. For the executable, select the dropdown and choose Select Other, then find the compiled CPython binary in the source directory. It will be called `python` or `python.exe`.
4. Enter any program arguments you wish to always have, such as `-X dev` to enable development mode. These flags are covered later in "Setting Runtime Configuration With the Command Line."
5. Set the working directory to the CLion macro `$ProjectFileDir$`:

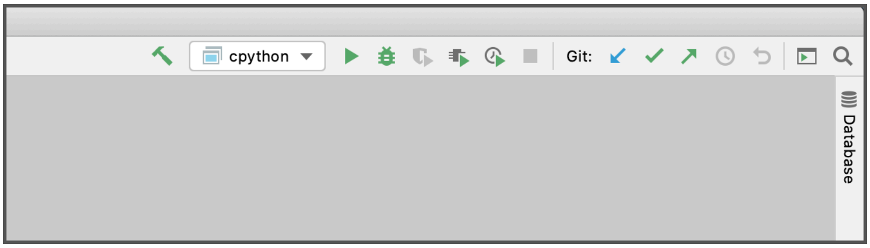


Name: cpython
Target: all
Executable: python.exe
Program arguments: -X dev
Working directory: $ProjectFileDir$
Environment variables:

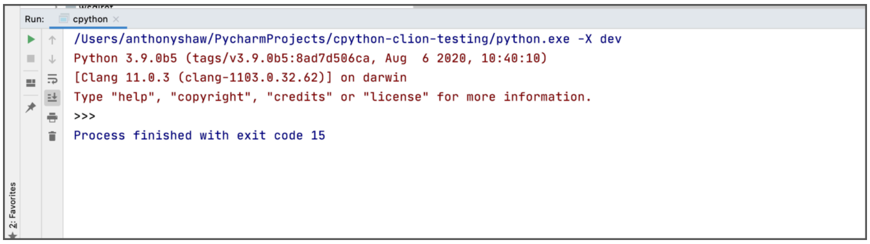
Click OK to add this configuration. You can repeat this step as many times as you like for any of the CPython `make` targets. See the section

"CPython's Make Targets" in the chapter "Compiling CPython" for a full reference.

The `cpython` build configuration will now be available in the top right of the CLion window:



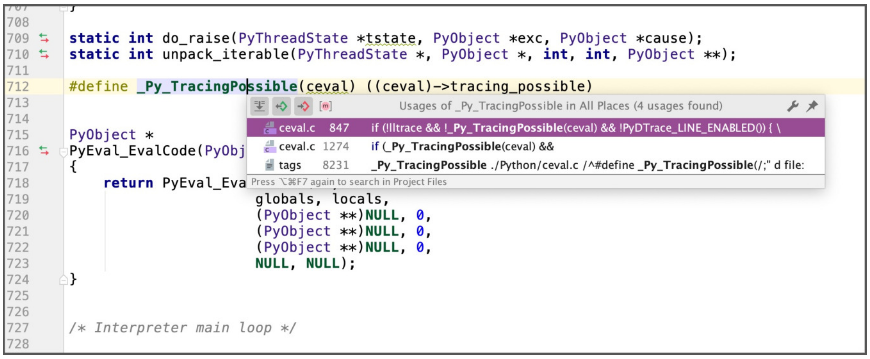
To test it out, click the arrow icon or select Run > Run 'cpython' from the top menu. You should now see the REPL at the bottom of the CLion window:



```
/Users/anthonyshaw/PycharmProjects/cpython-clion-testing/python.exe -X dev
Python 3.9.0b5 (tags/v3.9.0b5:8ad7d506ca, Aug  6 2020, 10:40:10)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
Process finished with exit code 15
```

Great! Now you can make changes and quickly try them out by clicking Build and Run. If you put any breakpoints in the C code, then make sure you choose Debug instead of Run.

Within the code editor, the shortcuts Cmd + Click on macOS and Ctrl + Click on Windows and Linux will bring up in-editor navigation features:



```
static int do_raise(PyThreadState *tstate, PyObject *exc, PyObject *cause);
static int unpack_iterable(PyThreadState *, PyObject *, int, int, PyObject **);

#define _Py_TracingPossible(ceval) ((ceval)->tracing_possible)


PyObject *
PyEval_EvalCode(PyObj
{
    return PyEval_Eva
                      globals, locals,
                      (PyObject **)NULL, 0,
                      (PyObject **)NULL, 0,
                      (PyObject **)NULL, 0,
                      NULL, NULL);
}


/* Interpreter main loop */
```

```
Usages of _Py_TracingPossible in All Places (4 usages found)
ceval.c   847   if (!lltrace && !_Py_TracingPossible(ceval) && !PyDTrace_LINE_ENABLED()) { \
ceval.c   1274  if (_Py_TracingPossible(ceval) &&
tags      8231  _Py_TracingPossible ./Python/ceval.c /^#define _Py_TracingPossible(/;" d file:
Press ⌥⌘F7 again to search in Project Files
```

Setting up Vim

Vim is a powerful console-based text editor. For fast development, use Vim with your hands resting on the keyboard home keys. The shortcuts and commands are within reach.

Note

On most Linux distributions and within the macOS Terminal, `vi` is an alias for `vim`. We'll use the `vim` command in this book, but if you have the alias, then `vi` will also work.

Out of the box, Vim has only basic functionality, little more than a text editor like Notepad. With some configuration and extensions, however, Vim can become a powerful tool for both Python and C editing.

Vim's extensions are in various locations, including GitHub. To ease the configuration and installation of plugins from GitHub, you can install a plugin manager like Vundle.

To install Vundle, run this command at the terminal:

```
$ git clone https://github.com/VundleVim/Vundle.vim.git \
  ~/.vim/bundle/Vundle.vim
```

Once Vundle is downloaded, you need to configure Vim to load the Vundle engine.

You'll install two plugins:

1. **Fugitive:** A status bar for Git with shortcuts for many Git tasks
2. **Tagbar:** A pane for making it easier to jump to functions, methods, and classes

To install these plugins, first change the contents of your Vim configuration file (normally `HOME▸.vimrc`) to include the following lines:

`cpython-book-samples▸11▸.vimrc`

```
syntax on
set nocompatible              " be iMproved, required
filetype off                  " required

" set the runtime path to include Vundle and initialize
set rtp+=~/.vim/bundle/Vundle.vim
call vundle#begin()

" let Vundle manage Vundle, required
Plugin 'VundleVim/Vundle.vim'

" The following are examples of different formats supported.
" Keep Plugin commands between vundle#begin/end.
" plugin on GitHub repo
Plugin 'tpope/vim-fugitive'
Plugin 'majutsushi/tagbar'
" All of your Plugins must be added before this line
call vundle#end()            " required
filetype plugin indent on    " required
" Open tagbar automatically in C files, optional
autocmd FileType c call tagbar#autoopen(0)
```

```
" Open tagbar automatically in Python files, optional
autocmd FileType python call tagbar#autoopen(0)
" Show status bar, optional
set laststatus=2
" Set status as git status (branch), optional
set statusline=%{FugitiveStatusline()}
```

To download and install these plugins, run the following command:

```
$ vim +PluginInstall +qall
```

You should see output for the download and installation of the plugins specified in the configuration file.

When editing or exploring the CPython source code, you will want to jump quickly between methods, functions, and macros. A basic text search won't distinguish a call to a function or its definition from the implementation. But you can use an application called ctags to index source files across a multitude of languages into a plain text database.

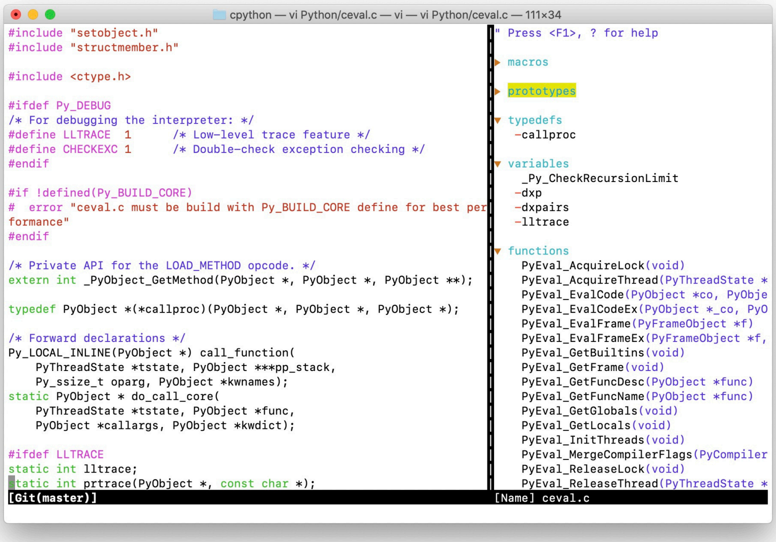
To index CPython's headers for all the C files and Python files in the standard library, run the following code:

```
$./configure
$ make tags
```

Now open the `Python▸ceval.c` file in Vim:

```
$ vim Python/ceval.c
```

You'll see the Git status at the bottom and the functions, macros, and variables in the right-hand pane:

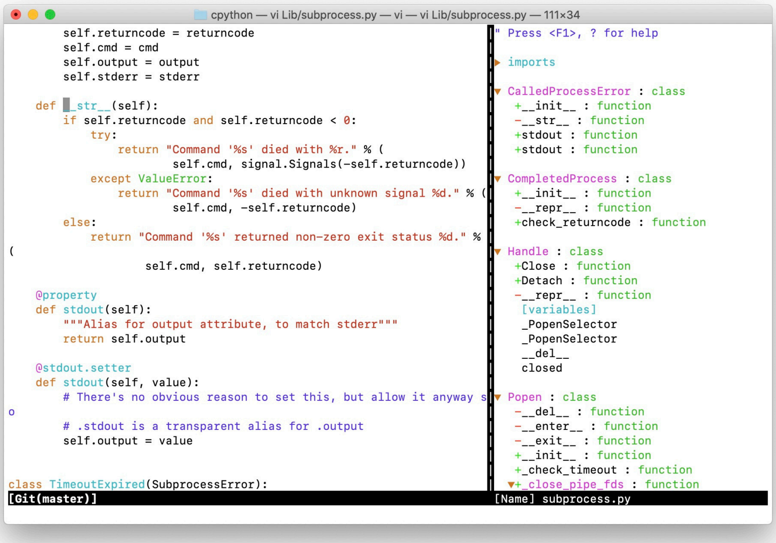

cpython — vi Python/ceval.c — vi — vi Python/ceval.c — 111×34

```
#include "setobject.h"                                              " Press <F1>, ? for help
#include "structmember.h"
                                                                     ▶ macros
#include <ctype.h>
                                                                     ▶ prototypes
#ifdef Py_DEBUG
/* For debugging the interpreter: */                                 ▼ typedefs
#define LLTRACE  1      /* Low-level trace feature */                   -callproc
#define CHECKEXC 1      /* Double-check exception checking */
#endif                                                               ▼ variables
                                                                         _Py_CheckRecursionLimit
#if !defined(Py_BUILD_CORE)                                             -dxp
#  error "ceval.c must be build with Py_BUILD_CORE define for best per  -dxpairs
formance"                                                               -lltrace
#endif
                                                                     ▼ functions
/* Private API for the LOAD_METHOD opcode. */                            PyEval_AcquireLock(void)
extern int _PyObject_GetMethod(PyObject *, PyObject *, PyObject **);     PyEval_AcquireThread(PyThreadState *
                                                                         PyEval_EvalCode(PyObject *co, PyObje
typedef PyObject *(*callproc)(PyObject *, PyObject *, PyObject *);       PyEval_EvalCodeEx(PyObject *_co, PyO
                                                                         PyEval_EvalFrame(PyFrameObject *f)
/* Forward declarations */                                               PyEval_EvalFrameEx(PyFrameObject *f,
Py_LOCAL_INLINE(PyObject *) call_function(                               PyEval_GetBuiltins(void)
    PyThreadState *tstate, PyObject ***pp_stack,                         PyEval_GetFrame(void)
    Py_ssize_t oparg, PyObject *kwnames);                                PyEval_GetFuncDesc(PyObject *func)
static PyObject * do_call_core(                                          PyEval_GetFuncName(PyObject *func)
    PyThreadState *tstate, PyObject *func,                               PyEval_GetGlobals(void)
    PyObject *callargs, PyObject *kwdict);                               PyEval_GetLocals(void)
                                                                         PyEval_InitThreads(void)
#ifdef LLTRACE                                                           PyEval_MergeCompilerFlags(PyCompiler
static int lltrace;                                                      PyEval_ReleaseLock(void)
static int prtrace(PyObject *, const char *);                            PyEval_ReleaseThread(PyThreadState *
[Git(master)]                                                        [Name] ceval.c
```

Next, open a Python file, such as `Lib` ▸ `subprocess.py`:

```
$ vim Lib/subprocess.py
```

Tagbar will show your imports, classes, methods, and functions:


cpython — vi Lib/subprocess.py — vi — vi Lib/subprocess.py — 111×34

```
        self.returncode = returncode
        self.cmd = cmd
        self.output = output
        self.stderr = stderr

    def __str__(self):
        if self.returncode and self.returncode < 0:
            try:
                return "Command '%s' died with %r." % (
                        self.cmd, signal.Signals(-self.returncode))
            except ValueError:
                return "Command '%s' died with unknown signal %d." % (
                        self.cmd, -self.returncode)
        else:
            return "Command '%s' returned non-zero exit status %d." %
(
                    self.cmd, self.returncode)

    @property
    def stdout(self):
        """Alias for output attribute, to match stderr"""
        return self.output

    @stdout.setter
    def stdout(self, value):
        # There's no obvious reason to set this, but allow it anyway s
o
        # .stdout is a transparent alias for .output
        self.output = value


class TimeoutExpired(SubprocessError):
[Git(master)]
```

```
" Press <F1>, ? for help

▸ imports

▾ CalledProcessError : class
   +__init__ : function
   -__str__ : function
   +stdout : function
   +stdout : function

▾ CompletedProcess : class
   +__init__ : function
   -__repr__ : function
   +check_returncode : function

▾ Handle : class
   +Close : function
   +Detach : function
   -__repr__ : function
    [variables]
    _PopenSelector
    _PopenSelector
    __del__
    closed

▾ Popen : class
   -__del__ : function
   -__enter__ : function
   -__exit__ : function
   +__init__ : function
   +_check_timeout : function
  ▾+_close_pipe_fds : function
[Name] subprocess.py
```

Within Vim, you can switch between windows with Ctrl + W, move to the right-hand pane with L, and use the arrow keys to move up and down between the tagged functions.

Press Enter to skip to any function implementation. To move back to the editor pane, press Ctrl + W, then press H.

See Also

Check out VIM Adventures for a fun way to learn and memorize the Vim commands.

Conclusion

If you're still undecided about which environment to use, then you don't need to make a decision right away. We used multiple environments while writing this book and working on changes to CPython.

Debugging is a critical feature for productivity, so having a reliable debugger that you can use to explore the runtime and understand bugs will save you a lot of time. If you're used to debugging in Python with `print()`, then it's important to note that this approach doesn't work in C. You'll cover debugging in full later in this book.

Compiling CPython

Now that you've downloaded a development environment and configured it, you can compile the CPython source code into an executable interpreter.

Unlike Python files, C source code must be recompiled each time it changes. You'll probably want to bookmark this chapter and memorize some of the steps, because you'll be repeating them a lot.

In the previous chapter, you saw how to set up your development environment with an option to run the build stage, which recompiles CPython. Before the build steps will work, you need a C compiler and some build tools.

The tools used depend on the operating system you're using, so skip ahead to the section for your operating system.

Note

If you're concerned that any of these steps will interfere with your existing CPython installations, don't worry. The CPython source directory behaves like a virtual environment.

When compiling CPython or modifying the source or the standard library, this all stays within the sandbox of the source directory.

If you want to install a custom version, this step is covered in this chapter.

Compiling CPython on macOS

Compiling CPython on macOS requires some additional applications and libraries. First, you'll need the essential C compiler tool kit. **Command Line Tools** is an app that you can update in macOS through the App Store. You need to perform the initial installation on the terminal.

Note

To open up a terminal in macOS, go to Applications > Other > Terminal. You'll want to save this app to your Dock, so Ctrl + Click the icon and select Keep in Dock.

Within the terminal, install the C compiler and tool kit by running the following:

```
$ xcode-select --install
```

After running this command, you'll be prompted to download and install a set of tools, including Git, Make, and the GNU C compiler.

You'll also need a working copy of OpenSSL to use for fetching packages from the PyPI website. If you plan on using this build to install additional packages, then SSL validation is required.

The most straightforward way to install OpenSSL on macOS is to use Homebrew.

Note

If you don't have Homebrew, then you can download and install it directly from GitHub with the following command:

```
$ /usr/bin/ruby -e "$(curl -fsSL \
  https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once you have Homebrew installed, you can install the dependencies for CPython with the `brew install` command:

```
$ brew install openssl xz zlib gdbm sqlite
```

Now that you have the dependencies, you can run the `configure` script.

The Homebrew command `brew --prefix <package>` will give the directory where `<package>` is installed. You will enable support for SSL by compiling the location that Homebrew uses.

The flag `--with-pydebug` enables debug hooks. Add this flag if you intend on debugging for development or testing purposes. Debugging CPython is covered extensively in the "Debugging" chapter.

The configuration stage needs to be run only once, with the location of the zlib package specified:

```
$ CPPFLAGS="-I$(brew --prefix zlib)/include" \
  LDFLAGS="-L$(brew --prefix zlib)/lib" \
  ./configure --with-openssl=$(brew --prefix openssl) \
  --with-pydebug
```

Running `./configure` will generate a makefile in the root of the repository. You can use it to automate the build process.

You can now build the CPython binary by running the following command:

```
$ make -j2 -s
```

See Also

For more information on the options for `make`, see the section "A Quick Primer on Make."

During the build, you may receive some errors. In the build summary, `make` will notify you that not all packages were built. For example, the `ossaudiodev`, `spwd`, and `_tkinter` packages will fail to build with this set of instructions. That's okay if you aren't planning on developing against these packages. If you are, then check out the Python Developer's Guide for more information.

The build will take a few minutes and generate a binary called `python.exe`. Every time you make changes to the source code, you'll need to rerun `make` with the same flags.

The `python.exe` binary is the debug binary of CPython. Execute `python.exe` to see a working REPL:

```
$ ./python.exe
Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Important

Yes, that's right, the macOS build has a `.exe` file extension. This extension is *not* because it's a Windows binary!

Because macOS has a case-insensitive file system, the developers didn't want people to accidentally refer to the directory `Python/` when working with the binary, so they appended `.exe` to avoid ambiguity.

If you later run `make install` or `make altinstall`, then the file will be renamed `python` before it's installed onto your system.

Compiling CPython on Linux

To compile CPython on Linux, you first need to download and install `make`, `gcc`, `configure`, and `pkgconfig`.

Use this command for Fedora Core, RHEL, CentOS, or other YUM-based systems:

```
$ sudo yum install yum-utils
```

Use this command for Debian, Ubuntu, or other APT-based systems:

```
$ sudo apt install build-essential
```

Then install some additional required packages.

Use this command for Fedora Core, RHEL, CentOS or other YUM-based systems:

```
$ sudo yum-builddep python3
```

Use this command for Debian, Ubuntu, or other APT-based systems:

```
$ sudo apt install libssl-dev zlib1g-dev libncurses5-dev \
  libncursesw5-dev libreadline-dev libsqlite3-dev libgdbm-dev \
  libdb5.3-dev libbz2-dev libexpat1-dev liblzma-dev libffi-dev
```

Now that you have the dependencies, you can run the `configure` script, optionally enabling the debug hooks using `--with-pydebug`:

```
$ ./configure --with-pydebug
```

Next, you can build the CPython binary by running the generated makefile:

```
$ make -j2 -s
```

See Also

For more help on the options for `make`, see the section "A Quick Primer on Make."

Review the output to ensure that there were no issues compiling the `_ssl` module. If there were, then check with your distribution for instructions on installing the headers for OpenSSL.

During the build, you may receive some errors. In the build summary, `make` will notify you that not all packages were built. That's okay if you aren't planning on developing against those packages. If you are, then check out the package details for required libraries.

The build will take a few minutes and generate a binary called `python`. This is the debug binary of CPython. Execute `./python` to see a working REPL:

```
$ ./python
Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
[Clang 10.0.1 (clang-1001.0.46.4)] on Linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Installing a Custom Version

If you're happy with your changes and want to use them inside your system, then you can install the Python binary from your source repository as a custom version.

For macOS and Linux, use the `altinstall` command, which doesn't create symbolic links for `python3` and installs a standalone version:

```
$ make altinstall
```

For Windows, you have to change the build configuration from Debug to Release, then copy the packaged binaries to a directory on your computer that is part of the system path.

A Quick Primer on Make

As a Python developer, you might not have come across `make` before. Or perhaps you have, but you haven't spent much time with it.

For C, C++, and other compiled languages, the list of commands you need to execute to load, link, and compile your code in the right order can be very long. When compiling applications from source, you need to link any external libraries in the system.

It would be unrealistic to expect the developer to know the locations of all of these libraries and to copy and paste them into the command line, so `make` and `configure` are commonly used in C/C++ projects to automate the creation of a build script.

When you executed `./configure`, `autoconf` searched your system for the libraries that CPython requires and copied their paths into a makefile.

The generated makefile is similar to a shell script and is broken into sections called **targets**.

Take the `docclean` target as an example. This target deletes some generated documentation files using the `rm` command:

```
docclean:
	-rm -rf Doc/build
	-rm -rf Doc/tools/sphinx Doc/tools/pygments Doc/tools/docutils
```

To execute this target, run `make docclean`. `docclean` is a simple target as it runs only two commands.

This is the convention for executing a `make` target:

```
$ make [options] [target]
```

If you call `make` without specifying a target, then `make` will run the default target, which is the first target specified in the makefile. For CPython, this is the `all` target, which compiles all parts of CPython.

`make` has many options. Here are some you'll find useful throughout this book:

Option	Use
`-d, --debug[=FLAGS]`	Print various types of debugging information
`-e, --environment-overrides`	Environment variables override makefiles
`-i, --ignore-errors`	Ignore errors from commands
`-j [N], --jobs[=N]`	Allow `N` jobs at once or infinite jobs otherwise
`-k, --keep-going`	Keep going when some targets can't be made
`-l [N], --load-average[=N], --max-load[=N]`	Start multiple jobs only if `load < N`

Option	Use
`-n`, `--dry-run`	Print commands instead of running them
`-s`, `--silent`	Don't echo commands
`-S`, `--stop`	Stop when targets can't be made

In the next section and throughout the book, you'll run `make` with these options:

```
$ make -j2 -s [target]
```

The `-j2` flag allows `make` to run two jobs simultaneously. If you have four or more cores, then you can change this to four or higher and the compilation will complete faster.

The `-s` flag stops the makefile from printing every command it runs to the console. If you want to see what's happening, then remove the `-s` flag.

CPython's Make Targets

For both Linux and macOS, you'll find yourself needing to clean up files, build, or refresh the configuration. The sections below contain tables outlining a number of useful `make` targets built into CPython's makefile.

Build Targets

The following targets are used for building the CPython binary:

Target	Purpose
`all` (default)	Build the compiler, libraries, and modules
`clinic`	Run Argument Clinic on all source files
`profile-opt`	Compile the Python binary with profile-guided optimization
`regen-all`	Regenerate all generated files
`sharedmods`	Build the shared modules

Test Targets

The following targets are used for testing your compiled binary:

Target	Purpose
`coverage`	Compile and run tests with `gcov`
`coverage-lcov`	Create coverage HTML reports
`quicktest`	Run a faster set of regression tests by excluding the tests that take a long time
`test`	Run a basic set of regression tests
`testall`	Run the full test suite twice, once without `.pyc` files and once with them
`testuniversal`	Run the test suite for both architectures in a universal build on OS X

Cleaning Targets

The primary cleaning targets are `clean`, `clobber`, and `distclean`. The `clean` target is for generally removing compiled and cached libraries and `.pyc` files.

If you find that `clean` doesn't do the job, then try `clobber`. The `clobber` target will remove your makefile, so you'll have to run `./configure` again.

To completely clean out an environment before distribution, run the `distclean` target.

The following list includes the three primary targets listed above, as well as some additional cleaning targets:

Target	Purpose
`check-clean-src`	Check that the source is clean when building out of source
`clean`	Remove `.pyc` files, compiled libraries, and profiles
`cleantest`	Remove `test_python_*` directories of previous failed test jobs
`clobber`	Same as `clean` but also remove libraries, tags, configurations, and builds

Target	Purpose
`distclean`	Same as `clobber` but also remove anything generated from source, such as makefiles
`docclean`	Remove built documentation in `Doc/`
`profile-removal`	Remove any optimization profiles
`pycremoval`	Remove `.pyc` files

Installation Targets

There are two flavors of installation targets: the default version, such as `install`, and the `alt` version, such as `altinstall`. If you want to install the compiled version onto your computer but don't want it to become the default Python 3 installation, then use the `alt` version of the commands:

Target	Purpose
`altbininstall`	Install the `python` interpreter with the version affixed, such as `python3.9`
`altinstall`	Install shared libraries, binaries, and documentation with the version suffix
`altmaninstall`	Install the versioned manuals
`bininstall`	Install all the binaries, such as `python`, `idle`, and `2to3`
`commoninstall`	Install shared libraries and modules
`install`	Install shared libraries, binaries, and documentation (will run `commoninstall`, `bininstall`, and `maninstall`)
`libinstall`	Install shared libraries
`maninstall`	Install the manuals
`sharedinstall`	Load modules dynamically

After you install with `make install`, the command `python3` will link to your compiled binary. If you use `make altinstall`, however, only `python$(VERSION)` will be installed, and the existing link for `python3` will remain intact.

Miscellaneous Targets

Below are some additional `make` targets that you may find useful:

Target	Purpose
autoconf	Regenerate configure and pyconfig.h.in
python-config	Generate the python-config script
recheck	Rerun configure with the same options as last time
smelly	Check that exported symbols start with Py or _Py (see PEP 7)
tags	Create a tags file for vi
TAGS	Create a tags file for Emacs

Compiling CPython on Windows

There are two ways to compile the CPython binaries and libraries from Windows:

1. Compile from the command prompt. This still requires the Microsoft Visual C++ compiler, which comes with Visual Studio.
2. Open PCbuild▸pcbuild.sln from Visual Studio and build directly.

In the sections below, you'll explore both of these options.

Installing the Dependencies

For both the command prompt compile script and the Visual Studio solution, you need to install several external tools, libraries, and C headers.

Inside the PCbuild folder is a .bat file that automates this process for you. Open a command prompt window inside PCbuild and execute PCbuild▸get_externals.bat:

```
> get_externals.bat
Using py -3.7 (found 3.7 with py.exe)
Fetching external libraries...
Fetching bzip2-1.0.6...
Fetching sqlite-3.28.0.0...
Fetching xz-5.2.2...
Fetching zlib-1.2.11...
Fetching external binaries...
Fetching openssl-bin-1.1.1d...
Fetching tcltk-8.6.9.0...
Finished.
```

Now you can compile from either the command prompt or Visual Studio.

Compiling From the Command Prompt

To compile from the command prompt, you need to select the CPU architecture you want to compile against. The default is win32, but chances are that you want a 64-bit (`amd64`) binary.

If you do any debugging, then the debug build comes with the ability to attach breakpoints in the source code. To enable the debug build, you add `-c Debug` to specify the debug configuration.

By default, `build.bat` will fetch external dependencies, but because we've already done that step, it will print a message skipping downloads:

```
> build.bat -p x64 -c Debug
```

This command will produce the Python binary `PCbuild` ▸ `amd64` ▸ `python_d.exe`. Start that binary directly from the command prompt:

```
> amd64\python_d.exe

Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
 [MSC v.1922 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You're now inside the REPL of your compiled CPython binary.

To compile a release binary, use this command:

```
> build.bat -p x64 -c Release
```

This command will produce the binary `PCbuild`▸`amd64`▸`python.exe`.

Note

The suffix `_d` specifies that CPython was built in the debug configuration.

The released binaries on Python.org are compiled in the profile-guided optimization (PGO) configuration. See the "Profile-Guided Optimization (PGO)" section at the end of this chapter for more details on PGO.

Arguments

The following arguments are available in `build.bat`:

Flag	Purpose	Expected Value
`-p`	Build platform CPU architecture	`x64`, `Win32` (default), `ARM`, `ARM64`
`-c`	Build configuration	`Release` (default), `Debug`, `PGInstrument` or `PGUpdate`
`-t`	Build target	`Build` (default), `Rebuild`, `Clean`, `CleanAll`

Flags

Here are some optional flags you can use for `build.bat`:

Flag	Purpose
`-v`	Verbose mode: show informational messages during build
`-vv`	Very verbose mode: show detailed messages during build
`-q`	Quiet mode: show only warnings and errors during build
`-e`	Download and install external dependencies (default)
`-E`	*Don't* download or install external dependencies
`--pgo`	Build with profile-guided optimization
`--regen`	Regenerate all grammar and tokens (used when you update the language)

For a full list, run `build.bat -h`.

Compiling From Visual Studio

Inside the `PCbuild` folder is a Visual Studio solution file, `PCbuild ▸ pcbuild.sln`, for building and exploring CPython source code.

When the solution file is loaded, it will prompt you to retarget the projects inside the solution to the version of the C/C++ compiler that you have installed. Visual Studio will also target the release of the Windows SDK that you have installed.

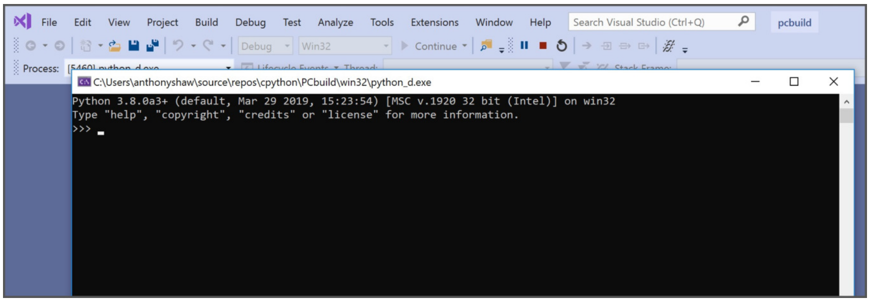
Be sure to change the Windows SDK version to the newest installed version and the platform toolset to the latest version. If you missed this window, then you can right-click the solution file in the Solutions and Projects window and select Retarget Solution.

Navigate to Build > Configuration Manager and ensure the Active Solution Configuration drop-down list is set to Debug and the Active Solution Platform list is set to either `x64` for 64-bit CPU architecture or `win32` for 32-bit.

Next, build CPython by pressing Ctrl + Shift + B or choosing Build > Build Solution. If you receive any errors about the Windows SDK being missing, make sure you set the right targeting settings in the Retarget Solution window. You should also see a Windows Kits folder in your Start menu with Windows Software Development Kit inside it.

The build stage could take ten minutes or more the first time. Once the build completes, you may see a few warnings that you can ignore.

To start the debug version of CPython, press F5, and CPython will launch the REPL in debug mode:


File
Edit
View
Project
Build
Debug
Test
Analyze
Tools
Extensions
Window
Help
Search Visual Studio (Ctrl+Q)
pcbuild
Debug
Win32
Continue
Process: [5460] python_d.exe
C:\Users\anthonyshaw\source\repos\cpython\PCbuild\win32\python_d.exe

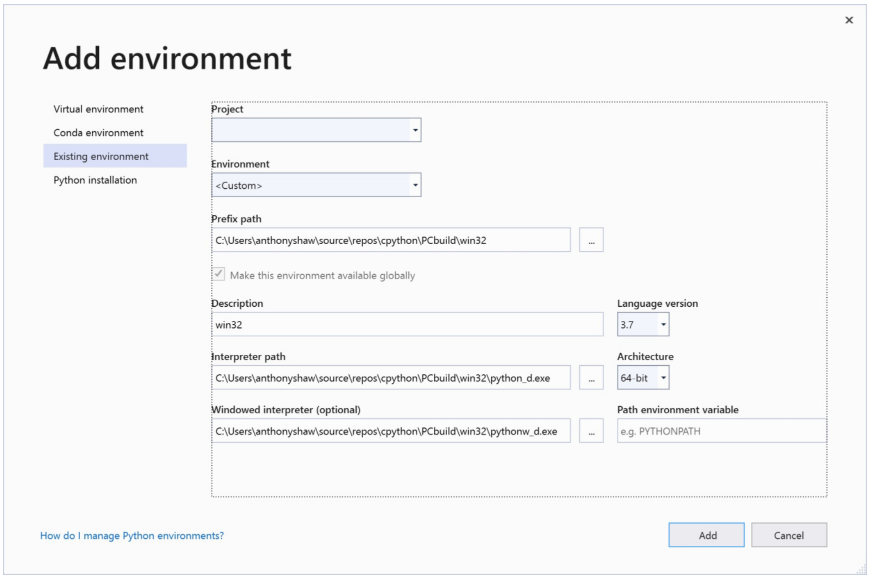
```
Python 3.8.0a3+ (default, Mar 29 2019, 15:23:54) [MSC v.1920 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

You can run the release build by changing the build configuration from Debug to Release on the top menu bar and rerunning Build › Build Solution. You now have both debug and release versions of the CPython binary within `PCbuild` ▸ `amd64`.

You can set up Visual Studio to be able to open a REPL with either the release or debug build by choosing Tools › Python › Python Environments from the top menu. In the Python Environments panel, click Add Environment and then target the debug or release binary. The debug binary will end in `_d.exe`, such as `python_d.exe` or `pythonw_d.exe`.

You'll most likely want to use the debug binary as it comes with debugging support in Visual Studio and will be useful as you read through this book.

In the Add Environment window, target the `python_d.exe` file as the interpreter inside `PCbuild` ▸ `amd64` and the `pythonw_d.exe` as the windowed interpreter:



Project:

Environment: <Custom>

Prefix path: C:\Users\anthonyshaw\source\repos\cpython\PCbuild\win32

✓ Make this environment available globally

Description: win32

Language version: 3.7

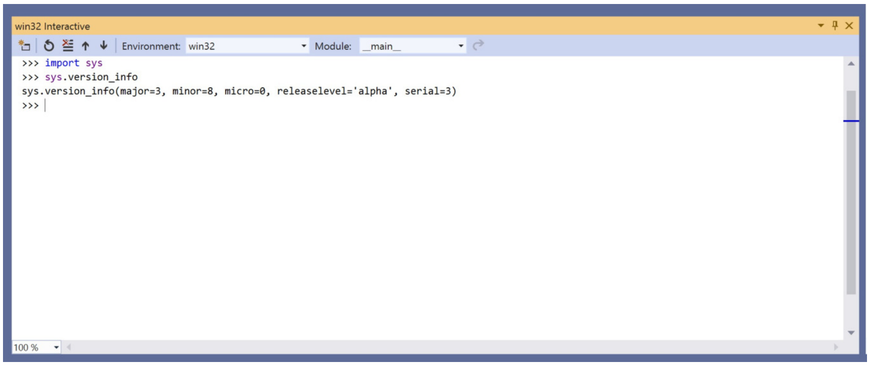
Interpreter path: C:\Users\anthonyshaw\source\repos\cpython\PCbuild\win32\python_d.exe

Architecture: 64-bit

Windowed interpreter (optional): C:\Users\anthonyshaw\source\repos\cpython\PCbuild\win32\pythonw_d.exe

Path environment variable: e.g. PYTHONPATH

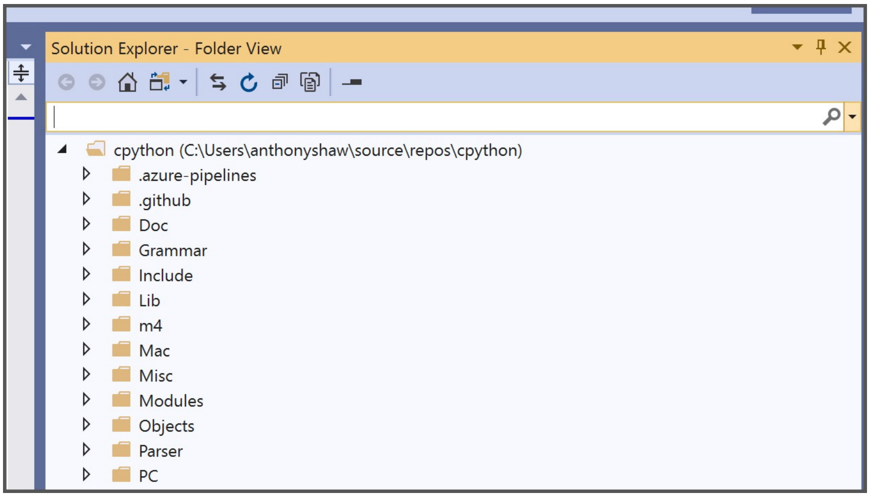
Start a REPL session by clicking `Open Interactive Window` in the Python Environments window and you'll see the REPL for the compiled version of Python:



```
>>> import sys
>>> sys.version_info
sys.version_info(major=3, minor=8, micro=0, releaselevel='alpha', serial=3)
>>>
```

Throughout this book, there will be REPL sessions with example commands. I encourage you to use the debug binary to run these REPL sessions in case you want to put in any breakpoints within the code.

To make it easier to navigate the code, in the Solution view, click the toggle button next to the Home icon to switch to Folder view:


Solution Explorer - Folder View
cpython (C:\Users\anthonyshaw\source\repos\cpython)
.azure-pipelines
.github
Doc
Grammar
Include
Lib
m4
Mac
Misc
Modules
Objects
Parser
PC

Profile-Guided Optimization

The macOS, Linux, and Windows build processes have flags for **profile-guided optimization (PGO)**. PGO isn't something created by the Python team, but a feature of many compilers, including those used by CPython.

PGO works by doing an initial compilation, then profiling the application by running a series of tests. The profile is then analyzed, and the compiler makes changes to the binary that improve performance.

For CPython, the profiling stage runs `python -m test --pgo`, which executes the regression tests specified in `Lib` ▸ `test` ▸ `libregrtest` ▸ `pgo.py`. These tests have been specifically selected because they use a commonly used C extension module or type.

Note

The PGO process is time-consuming, so to keep your compilation time short, I've excluded it from the lists of recommended steps offered throughout this book.

If you want to distribute a custom-compiled version of CPython into a production environment, then you should run `./configure` with the `--with-pgo` flag in Linux and macOS and use the `--pgo` flag in `build.bat` on Windows.

Because the optimizations are specific to the platform and architecture that the profile was executed on, PGO profiles can't be shared between operating systems or CPU architectures. The distributions of CPython on Python.org have already been through PGO, so if you run a benchmark on a vanilla-compiled binary, then it will be slower than one downloaded from Python.org.

The Windows, macOS, and Linux profile-guided optimizations include these checks and improvements:

- **Function inlining**: If a function is regularly called from another function, then it will be **inlined**, or copied into the calling function, to reduce the stack size.
- **Virtual call speculation and inlining**: If a virtual function call frequently targets a certain function, then PGO can insert a conditionally executed direct call to that function. The direct call can then be inlined.
- **Register allocation optimization**: Based on profile data results, the PGO will optimize register allocation.
- **Basic block optimization**: Basic block optimization allows commonly executed basic blocks that temporally execute within a given frame to be placed in the same **locality**, or set of pages. It minimizes the number of pages used, which minimizes memory overhead.

- **Hot spot optimization**: Functions that the program spends the most execution time on can be optimized for speed.
- **Function layout optimization**: After PGO analyzes the call graph, functions that tend to be along the same execution path are moved to the same section of the compiled application.
- **Conditional branch optimization**: PGO can look at a decision branch, like an `if` ... `else if` or `switch` statement, and spot the most commonly used path. For example, if there are ten cases in a `switch` statement, and one is used 95 percent of the time, then that case will be moved to the top so that it will be executed immediately in the code path.
- **Dead spot separation**: Code that isn't called during PGO is moved to a separate section of the application.

Conclusion

In this chapter, you've seen how to compile CPython source code into a working interpreter. You'll use this knowledge throughout the book as you explore and adapt the source code.

You might need to repeat the compilation steps dozens or even hundreds of times when working with CPython. If you can adapt your development environment to create shortcuts for recompilation, then it's better to do that now and save yourself a lot of time.

The Python Language and Grammar

The purpose of a compiler is to convert one language into another. Think of a compiler like a translator. You would hire a translator to listen to you speaking in English and then repeat your words in a different language, like Japanese.

To accomplish this, the translator must understand the grammatical structures of both the source and target languages.

Some compilers will compile into a low-level machine code that can be executed directly on a system. Other compilers will compile into an intermediary language to be executed by a virtual machine.

One consideration when choosing a compiler is the system portability requirements. Java and .NET CLR will compile into an intermediary language so that the compiled code is portable across multiple system architectures. C, Go, C++, and Pascal will compile into an executable binary. This binary is built for the platform on which it was compiled.

Python applications are typically distributed as source code. The role of the Python interpreter is to convert the Python source code and execute it in one step. The CPython runtime compiles your code when it runs for the first time. This step is invisible to the regular user.

Python code isn't compiled into machine code. It's compiled into a low-level intermediary language called **bytecode**. This bytecode is stored in `.pyc` files and cached for execution. If you run the same

Python application twice without changing the source code, then it will be faster on the second execution. This is because it loads the compiled bytecode instead of recompiling each time.

Why CPython Is Written in C and Not Python

The **C** in CPython is a reference to the C programming language, indicating that this Python distribution is written in the C language.

This statement is mostly true. The compiler in CPython is written in pure C. However, many of the standard library modules are written in pure Python or a combination of C and Python.

So Why Is the CPython Compiler Written in C and Not Python?

The answer is based on how compilers work. There are two types of compilers:

1. **Self-hosted compilers** are compilers written in the language they compile, such as the Go compiler. This is done by a process known as **bootstrapping**.
2. **Source-to-source compilers** are compilers written in another language that already has a compiler.

If you're writing a new programming language from scratch, then you need an executable application to compile your compiler! You need a compiler to execute anything, so when new languages are developed, they're often written first in an older, more established language.

There are also tools available that can take a language specification and create a parser, which you'll learn about later in this chapter. Popular compiler-compilers include GNU Bison, Yacc, and ANTLR.

See Also

If you want to learn more about parsers, then check out the Lark project. Lark is a parser for context-free grammar written in Python.

An excellent example of compiler bootstrapping is the Go programming language. The first Go compiler was written in C, then once Go could be compiled, the compiler was rewritten in Go.

CPython, on the other hand, kept its C heritage. Many of the standard library modules, like the `ssl` module or the `sockets` module, are written in C to access low-level operating system APIs.

The APIs in the Windows and Linux kernels for creating network sockets, working with the file system, or interacting with the display were all written in C, so it made sense for Python's extensibility layer to be focused on the C language. Later in this book, you'll cover the Python standard library and the C modules.

There is a Python compiler written in Python called PyPy. PyPy's logo is an Ouroboros to represent the self-hosting nature of the compiler.

Another example of a cross-compiler for Python is Jython. Jython is written in Java and compiles from Python source code into Java bytecode. In the same way that CPython makes it easy to import C libraries and use them from Python, Jython makes it easy to import and reference Java modules and classes.

The first step to creating a compiler is to define the language. For example, this is not valid Python:

```
def my_example() <str> :
{
    void* result = ;
}
```

The compiler needs strict rules for the grammatical structure for the language before it tries to execute it.

Note

For the rest of this book, `./python` will refer to the compiled version of CPython. However, the actual command will depend on your operating system.

For Windows:

```
> python.exe
```

For Linux:

```
$ ./python
```

For macOS:

```
$ ./python.exe
```

The Python Language Specification

Contained within the CPython source code is the definition of the Python language. This document is the reference specification used by all the Python interpreters.

The specification is in both a human-readable and a machine-readable format. Inside the documentation is a detailed explanation of the Python language outlining what is allowed and how each statement should behave.

Language Documentation

The `Doc` ▸ `reference` directory contains reStructuredText explanations of the features in the Python language. These files form the official Python reference guide at docs.python.org/3/reference.

Inside the directory are the files you need to understand the whole language, structure, and keywords:

cpython/Doc/reference

compound_stmts.rst	Compound statements like `if`, `while`, `for`, and function definitions
datamodel.rst	Objects, values, and types
executionmodel.rst	The structure of Python programs
expressions.rst	The elements of Python expressions
grammar.rst	Python's core grammar (referencing Grammar/Grammar)
import.rst	The import system
index.rst	Index for the language reference
introduction.rst	Introduction to the reference documentation
lexical_analysis.rst	Lexical structure like lines, indentation, tokens, and keywords
simple_stmts.rst	Simple statements like `assert`, `import`, `return`, and `yield`
toplevel_components.rst	Description of the ways to execute Python, like scripts and modules

An Example

Inside `Doc` ▸ `reference` ▸ `compound_stmts.rst`, you can see a simple example defining the `with` statement.

The `with` statement has many forms, the simplest being the instantiation of a context manager and a nested block of code:

```
with x():
   ...
```

You can assign the result to a variable using the `as` keyword:

```
with x() as y:
   ...
```

You can also chain context managers together with a comma:

```
with x() as y, z() as jk:
   ...
```

The documentation contains the human-readable specification of the language. The machine-readable specification is housed in a single file, `Grammar` ▸ `python.gram`.

The Grammar File

Python's grammar file uses a parsing expression grammar (PEG) specification. In the grammar file you can use the following notation:

- `*` for repetition
- `+` for at-least-once repetition
- `[]` for optional parts
- `|` for alternatives
- `()` for grouping

As an example, think about how you would define a cup of coffee:

- It must have a cup.
- It must include at least one shot of espresso and can contain multiple shots.
- It can have milk, but this is optional.
- It can have water, but this is optional.
- If it contains milk, then the milk can be of various types, like full-fat, skimmed, or soy.

Defined in PEG, a coffee order could look like this:

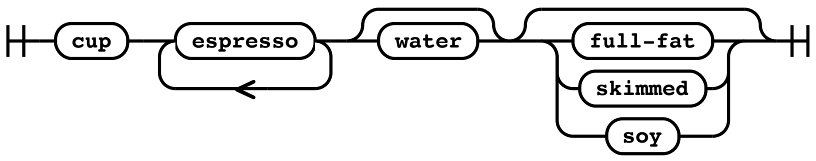
```
coffee: 'cup' ('espresso')+ ['water'] [milk]
milk: 'full-fat' | 'skimmed' | 'soy'
```

See Also

In CPython 3.9, the CPython source code has two grammar files. One legacy grammar is written in a context-free notation called Backus-Naur Form (BNF). In CPython 3.10, the BNF grammar file (`Grammar` ▸ `Grammar`) has been removed.

BNF isn't specific to Python and is often used as the notation for grammar in many other languages.

In this chapter, you'll visualize grammar with railroad diagrams. Here's a railroad diagram for the coffee statement:



In a railroad diagram, each possible combination must go in a line from left to right. Optional statements can be bypassed, and some statements can be formed as loops.

Example: `while` Statement

There are a few forms of the `while` statement. The simplest contains an expression, then the `:` terminal followed by a block of code:

```
while finished == True:
    do_things()
```

Alternatively, you can use an assignment expression, which is referred to in the grammar as a `named_expression`. This is a new feature as of Python 3.8:

```
while letters := read(document, 10):
    print(letters)
```

Optionally, `while` statements can be followed by an `else` statement and block:

```
while item := next(iterable):
    print(item)
else:
    print("Iterable is empty")
```

If you search for `while_stmt` in the grammar file, then you can see the definition:

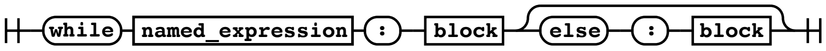
```
while_stmt[stmt_ty]:
    | 'while' a=named_expression ':' b=block c=[else_block] ...
```

Anything in quotes is a string literal, known as a **terminal**. Terminals are how keywords are recognized.

There are references to two other definitions in these two lines:

1. **`block`** refers to a block of code with one or multiple statements.
2. **`named_expression`** refers to a simple expression or assignment expression.

Visualized in a railroad diagram, the `while` statement looks like this:



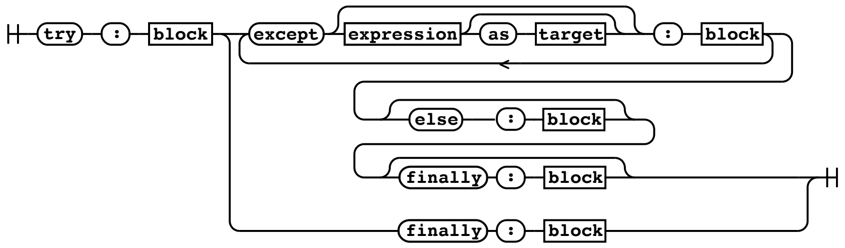
As a more complex example, the `try` statement is defined in the grammar like this:

```
try_stmt[stmt_ty]:
    | 'try' ':' b=block f=finally_block { _Py_Try(b, NULL, NULL, f, EXTRA) }
    | 'try' ':' b=block ex=except_block+ el=[else_block] f=[finally_block]..
except_block[excepthandler_ty]:
    | 'except' e=expression t=['as' z=target { z }] ':' b=block {
        _Py_ExceptHandler(e, (t) ? ((expr_ty) t)->v.Name.id : NULL, b,  ...
    | 'except' ':' b=block { _Py_ExceptHandler(NULL, NULL, b, EXTRA) }
finally_block[asdl_seq*]: 'finally' ':' a=block { a }
```

There are two uses of the `try` statement:

1. `try` with only a `finally` statement
2. `try` with one or many `except` clauses, followed by an optional `else`, then an optional `finally`

Here are those same options visualized in a railroad diagram:



The `try` statement is a good example of a more complex structure.

If you want to understand the Python language in detail, then read through the grammar defined in `Grammar` ▸ `python.gram`.

The Parser Generator

The grammar file itself is never used by the Python compiler. Instead, a parser generator reads the file and generates a parser. If you make changes to the grammar file, then you must regenerate the parser and recompile CPython.

The CPython parser was rewritten in Python 3.9 from a parser table automaton (the `pgen` module) into a contextual grammar parser.

In Python 3.9, the old parser is available at the command line by using the `-X oldparser` flag, and in Python 3.10 it's removed completely. This book refers to the new parser implemented in 3.9.

Regenerating Grammar

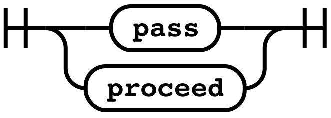
To see `pegen`, the new PEG generator introduced in CPython 3.9, in action, you can change part of the Python grammar. Search `Grammar` ▸ `python.gram` for `small_stmt` to see the definition of small statements:

```
small_stmt[stmt_ty] (memo):
    | assignment
    | e=star_expressions { _Py_Expr(e, EXTRA) }
    | &'return' return_stmt
    | &('import' | 'from') import_stmt
    | &'raise' raise_stmt
    | 'pass' { _Py_Pass(EXTRA) }
    | &'del' del_stmt
    | &'yield' yield_stmt
    | &'assert' assert_stmt
    | 'break' { _Py_Break(EXTRA) }
    | 'continue' { _Py_Continue(EXTRA) }
    | &'global' global_stmt
    | &'nonlocal' nonlocal_stmt
```

In particular, the line `'pass' { _Py_Pass(EXTRA) }` is for the `pass` statement:

Change that line to accept the terminal (keyword) `'pass'` or `'proceed'` as keywords by adding a choice, `|`, and the `'proceed'` literal:

```
    | ('pass'|'proceed') { _Py_Pass(EXTRA) }
```



Next, rebuild the grammar files. CPython comes with scripts to automate grammar regeneration.

On macOS and Linux, run the `make regen-pegen` target:

```
$ make regen-pegen
```

For Windows, bring up a command prompt from the `PCBuild` directory and run `build.bat` with the `--regen` flag:

```
> build.bat --regen
```

You should see an output showing that the new `Parser▸pegen▸parse.c` file has been regenerated.

With the regenerated parser table, when you recompile CPython, it will use the new syntax. Use the same compilation steps you used for your operating system in the last chapter.

If the code compiled successfully, then you can execute your new CPython binary and start a REPL.

In the REPL, you can now try defining a function. Instead of using the `pass` statement, use the `proceed` keyword alternative that you compiled into the Python grammar:

```
$ ./python

Python 3.9 (tags/v3.9:9cf67522, Oct 5 2020, 10:00:00)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def example():
...    proceed
...
>>> example()
```

Congratulations, you've changed the CPython syntax and compiled your own version of CPython!

Next, you'll explore tokens and their relationship to grammar.

Tokens

Alongside the grammar file in the `Grammar` folder is the `Grammar` ▸ `Tokens` file, which contains each of the unique types found as leaf nodes in a parse tree. Each token also has a name and a generated unique ID. The names make it simpler to refer to tokens in the tokenizer.

Note

The `Grammar` ▸ `Tokens` file is a new feature in Python 3.8.

For example, the left parenthesis is called `LPAR`, and semicolons are called `SEMI`. You'll see these tokens later in the book:

```
LPAR                    '('
RPAR                    ')'
LSQB                    '['
RSQB                    ']'
COLON                   ':'
COMMA                   ','
SEMI                    ';'
```

As with the `Grammar` file, if you change the `Grammar` ▸ `Tokens` file, you need to rerun `pegen`.

To see tokens in action, you can use the `tokenize` module in CPython.

Note

The tokenizer written in Python is a utility module. The actual Python parser uses a different process for identifying tokens.

Create a simple Python script called `test_tokens.py`:

`cpython-book-samples` ▸ `13` ▸ `test_tokens.py`

```python
# Demo application
def my_function():
   proceed
```

Input the `test_tokens.py` file to a module built into the standard library called `tokenize`. You'll see the list of tokens by line and character. Use the `-e` flag to output the exact token names:

```
$ ./python -m tokenize -e test_tokens.py

0,0-0,0:            ENCODING       'utf-8'
1,0-1,14:           COMMENT        '# Demo application'
1,14-1,15:          NL             '\n'
2,0-2,3:            NAME           'def'
2,4-2,15:           NAME           'my_function'
2,15-2,16:          LPAR           '('
2,16-2,17:          RPAR           ')'
2,17-2,18:          COLON          ':'
2,18-2,19:          NEWLINE        '\n'
3,0-3,3:            INDENT         '   '
3,3-3,7:            NAME           'proceed'
3,7-3,8:            NEWLINE        '\n'
4,0-4,0:            DEDENT         ''
4,0-4,0:            ENDMARKER      ''
```

In the output, the first column is the range of the line and column coordinates, the second column is the name of the token, and the final column is the value of the token.

In the output, the `tokenize` module has implied some tokens:

- The `ENCODING` token for `utf-8`
- A `DEDENT` to close the function declaration
- An `ENDMARKER` to end the file
- A blank line at the end

It's best practice to have a blank line at the end of your Python source files. If you omit it, then CPython adds one for you.

The `tokenize` module is written in pure Python and is located in `Lib ▸ tokenize.py`.

To see a verbose readout of the C parser, you can run a debug build of Python with the `-d` flag. Using the `test_tokens.py` script you created earlier, run it with the following:

```
$ ./python -d test_tokens.py

 > file[0-0]: statements? $
  > statements[0-0]: statement+
   > _loop1_11[0-0]: statement
    > statement[0-0]: compound_stmt
 ...
  + statements[0-10]: statement+ succeeded!
 + file[0-11]: statements? $ succeeded!
```

In the output, you can see that it highlighted `proceed` as a keyword. In the next chapter, you'll see how executing the Python binary gets to the tokenizer and what happens from there to execute your code.

To clean up your code, revert the change in `Grammar` ▸ `python.gram`, regenerate the grammar again, then clean the build and recompile.

Use the following for macOS or Linux:

```
$ git checkout -- Grammar/python.gram
$ make regen-pegen
$ make -j2 -s
```

Or use the following for Windows:

```
> git checkout -- Grammar/python.gram
> build.bat --regen
> build.bat -t CleanAll
> build.bat -t Build
```

Conclusion

In this chapter, you've been introduced to the Python grammar definitions and parser generator. In the next chapter, you'll expand on that knowledge to build a more complex syntax feature, an "almost-equal"