

CPYTHON INTERNAL



**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

Copyright © Real Python (realpython.com), 2012–2020

For online information and ordering of this and other books by Real Python, please visit realpython.com. For more information, please contact us at info@realpython.com.

ISBN: 9781775093343 (paperback)

ISBN: 9781775093350 (electronic)

Cover design by Aldren Santos

“Python” and the Python logos are trademarks or registered trademarks of the Python Software Foundation, used by Real Python with permission from the Foundation.

Thank you for downloading this ebook. This ebook is licensed for your personal enjoyment only. This ebook may not be re-sold or given away to other people. If you would like to share this book with another person, please purchase an additional copy for each recipient. If you’re reading this book and did not purchase it, or it was not purchased for your use only, then please return to realpython.com/cpython-internals and purchase your own copy. Thank you for respecting the hard work behind this book.

Updated 2020-05-29 We would like to thank our early access readers for their excellent feedback: Jim Anderson, Michal Porteš, Dan Bader, Florian Dahlitz, Mateusz Stawiarski, Evance Soumaoro, Fletcher Graham, André Roberge, Daniel Hao, Kimia. Thank you all!

This is an Early Access version of “CPython Internals: Your Guide to the Python 3 Interpreter”

With your help we can make this book even better:

At the end of each section of the book you’ll find a “magical” feedback link. Clicking the link takes you to an **online feedback form** where you can share your thoughts with us.

Please feel free to be as terse or detailed as you see fit. All feedback is stored anonymously, but you can choose to leave your name and contact information so we can follow up or mention you on our “Thank You” page.

We use a different feedback link for each section, so we’ll always know which part of the book your notes refer to.

Thank you for helping us make this book an even more valuable learning resource for the Python community.

— Anthony Shaw

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

“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

“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 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

Acknowledgements

Thank you to my wife, Verity, for her support and patience. Without her this wouldn't be possible.

Thank you to everyone who has supported me on this journey.

– Anthony Shaw

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.

Contents

Contents	8
Foreword	13
Introduction	15
How to Use This Book	16
Bonus Material & Learning Resources	17
Getting the CPython Source Code	21
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	52
Profile Guided Optimization	58
Conclusion	60

The Python Language and Grammar	61
Why CPython Is Written in C and Not Python	62
The Python Language Specification	64
Using the Parser Generator	69
The Parser Generator	69
Regenerating Grammar	70
A More Complex Example	75
Conclusion	78
Configuration and Input	80
Configuration State	83
Build Configuration	86
Building a Module From Input	87
Conclusion	93
Lexing and Parsing with Syntax Trees	94
Concrete Syntax Tree Generation	95
The CPython Parser-Tokenizer	98
Abstract Syntax Trees	103
Important Terms to Remember	113
Example: Adding an Almost Equal Comparison Operator	113
Conclusion	118
The Compiler	119
Related Source Files	121
Important Terms	121
Instantiating a Compiler	122
Future Flags and Compiler Flags	123
Symbol Tables	125
Core Compilation Process	132
Assembly	138
Creating a Code Object	143
Using Instaviz to Show a Code Object	144
Example: Implementing the “Almost-Equal” Operator	147
Conclusion	152

The Evaluation Loop	154
Stack Frames	155
Related Source Files	156
Important Terms	156
Constructing Thread State	156
Constructing Frame Objects	158
Frame Execution	166
The Value Stack	169
Example: Adding an Item to a List	175
Conclusion	180
Memory Management	182
Memory Allocation in C	182
Design of the Python Memory Management System	186
The CPython Memory Allocator	188
The Object and PyMem Memory Allocation Domains	198
The Raw Memory Allocation Domain	202
Custom Domain Allocators	202
Custom Memory Allocation Sanitizers	203
The PyArena Memory Arena	206
Reference Counting	207
Garbage Collection	214
Conclusion	224
Parallelism and Concurrency	226
Models of Parallelism and Concurrency	228
The Structure of a Process	228
Multi-Process Parallelism	231
Multithreading	255
Asynchronous Programming	268
Generators	269
Coroutines	275
Asynchronous Generators	281
Subinterpreters	282
Conclusion	286

Objects and Types	288
Examples in This Chapter	289
Builtin Types	290
Object and Variable Object Types	291
The <code>type</code> Type	292
Bool and Long Integer Type	296
Unicode String Type	301
Dictionary Type	311
Conclusion	316
The Standard Library	318
Python Modules	318
Python and C Modules	320
The Test Suite	324
Running the Test Suite on Windows	324
Running the Test Suite on Linux/macOS	325
Test Flags	326
Running Specific Tests	326
Testing Modules	328
Test Utilities	329
Conclusion	330
Debugging	331
Using the Crash Handler	332
Compiling Debug Support	332
Using Lldb for macOS	333
Using Gdb	337
Using Visual Studio Debugger	340
Using CLion Debugger	342
Conclusion	352

Benchmarking, Profiling, and Tracing	353
Using Timeit for Micro-Benchmarks	354
Using the Python Benchmark Suite for Runtime Benchmarks	356
Profiling Python Code with cProfile	362
Profiling C Code with Dtrace	365
Conclusion	370
Conclusion	371
Writing C Extensions for CPython	371
Using This Knowledge to Improve Your Python Applications	372
Using This Knowledge to Contribute to the CPython Project	373
Keep Learning	376
Appendix 1 : Introduction to C for Python Programmers	378
C Preprocessor	378
Basic C Syntax	381
Conclusion	389

Foreword

“A programming language created by a community fosters happiness in its users around the world.”

— Guido van Rossum, [King’s Day Speech 2016](#)

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 decades” 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 progressed from mainframes to cell phone service to the wide-ranging wonders of the web and cloud computing. All of 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 through many failures together built a solid foundation for success and growth.

CPython Internals will take you on a journey to explore the wildly successful language, **Python**. The book serves as a guidebook for learning how CPython is created under the hood. It will give you a glimpse of how the core developers crafted the language. Python’s strengths

include its readability and a 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 after reading this book your skills will grow and you will be able solve even more complex problems that can improve our world.

It’s my hope that Anthony motivates you to learn more about Python, inspire you to build innovative things, and give 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 & Member of the CPython Steering Council

Introduction

Are there certain parts of Python that just seem magic? Like how dictionaries are so much faster than looping over a list to find an item. How does a generator remember the state of the variables each time it yields a value, and why do you never have to allocate memory like other languages? It turns out, 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 cross-platform and straightforward. 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 need to understand how CPython works. These abstractions are not perfect, and they are leaky.

Once you understand how CPython works, you can optimize your applications and fully leverage its power. 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 concepts 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 for each chapter and make sure you try out the demos and the interactive elements. You can feel a sense of achievement that you grasp the core concepts of Python that can 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 in the book using 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 seeing what effect it has.

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

How skilled do I need to be in Python 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 the book.

Do I need to know C to use this book? You do not need to be proficient in C to use this book. If you are new to C, check out *Appendix 1: Introduction to C for Python Programmers* at the back of this book for a quick introduction.

How long will it take to finish this book? I don't recommend rushing this book, try reading a 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 over 20 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, whilst writing this book, I discovered many lines of code written by Guido van Rossum (the author of Python) and untouched since version 1.

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

Some of the concepts in this book are brand-new; some are even experimental. While writing this book, I came across issues in the source code and bugs in CPython. Then, they got **fixed or improved**. That's part of the wonder of CPython as a flourishing open-source project.

Bonus Material & Learning Resources

Online 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
```

```
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 that you're welcome to use any portion of the code for any purpose in your own programs.

Cython is licensed under the [PSF 2 license](#). Snippets and samples of Cython source code used in this book are done so under the license of the PSF 2.0 license terms.

Note

The code found in this book has been tested with Python 3.9.0b1 on Windows 10, macOS 10.15, and Linux.

Formatting Conventions

Code blocks will be 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.

Bold text will be used to denote a new or important term.

Notes and Warning 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 a warning 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 & 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](https://realpython.com), 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 two 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](https://twitter.com/realpython)
- The Real Python Email Newsletter

[Leave feedback on this section »](#)

Getting the CPython Source Code

When you type `python` at the console or install a Python distribution from python.org, you are running **CPython**. CPython is one of many Python implementations, maintained and written by different teams of developers. Some [alternatives](#) you may have heard 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 what you expect to be inside the Python distribution:

- When you type `python` without a file or module, it gives an interactive prompt (REPL).
- You can import built-in modules from the standard library like `json`, `csv`, and `collections`.
- 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. There’s a lot more than

just a compiler.

In this book, we'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?

Note

This book targets version [3.9.0b1](#) of the CPython source code.

The CPython source distribution comes with a whole range of tools, libraries, and components. We'll explore those in this book.

To download a copy of the CPython source code, you can use [git](#) to pull the latest version:

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

We are using version 3.9.0b1 throughout this book. Check out that version and create a local branch from it:

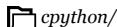
```
$ git checkout tags/v3.9.0b1 -b v3.9.0b1
```

Note

If you don't have Git available, you can install it from git-scm.com. Alternatively, you can download the CPython source in a [ZIP](#) file directly from the GitHub website.

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

Inside of the newly downloaded `cpython` directory, you will find the following subdirectories:



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

Next, you can set up your environment ready for development.

[Leave feedback on this section »](#)

Setting up Your Development Environment

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

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

IDE or Editor?

If you haven't yet decided which development environment to use, there is one decision to make first, whether to use an Integrated Development Environment (IDE) or 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 8GB), a code editor is recommended. IDEs also take longer to start-up. If you want to edit a file quickly, a code editor is a better choice.

There are 100's of editors and IDEs available for free or at a cost, here are some commonly used IDEs and Editors that would suit CPython development:

Application	Style	Supports
Microsoft VS 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 Visual Studio is available for Mac, but does not support Python Tools for Visual Studio, nor C compilation.

To aid the development of CPython, you will explore the setup steps for:

- 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 builtin support for Python and the C source code on Windows. I recommend it for this book. If you already have Visual Studio 2017 installed, that would also work.

Note

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

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. The bare minimum for this book is:

- The **Python Development** workload
- The optional **Python native development tools**
- Python 3 64-bit (3.7.2) (can be deselected if you already have Python 3.7 installed)

Deselect any other optional features if you want to be more conscientious with disk space.

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

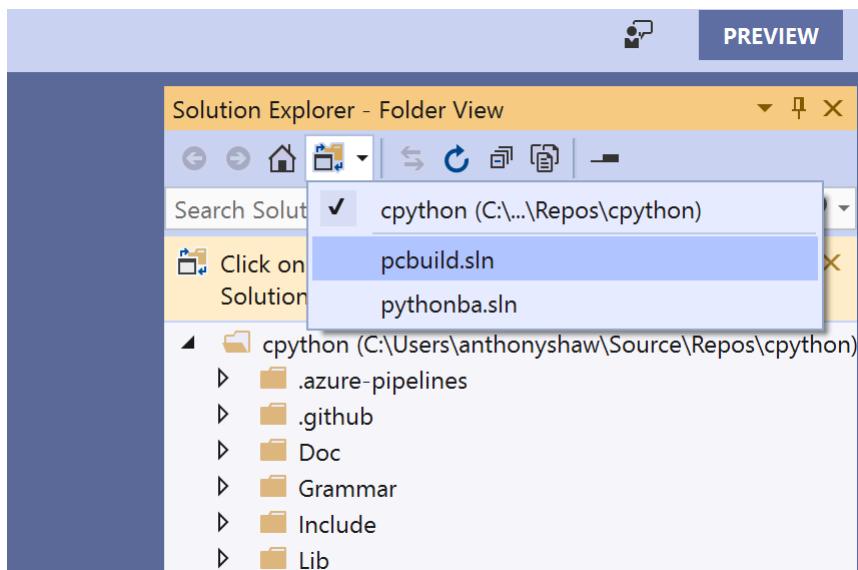
Once the installer has completed, click the **Launch** button to start Visual Studio. You will be prompted to sign in. If you have a Microsoft account, you can log in, or skip that step.

You will now 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>, chose 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 10 minutes.

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



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 on-line marketplace of plugins.

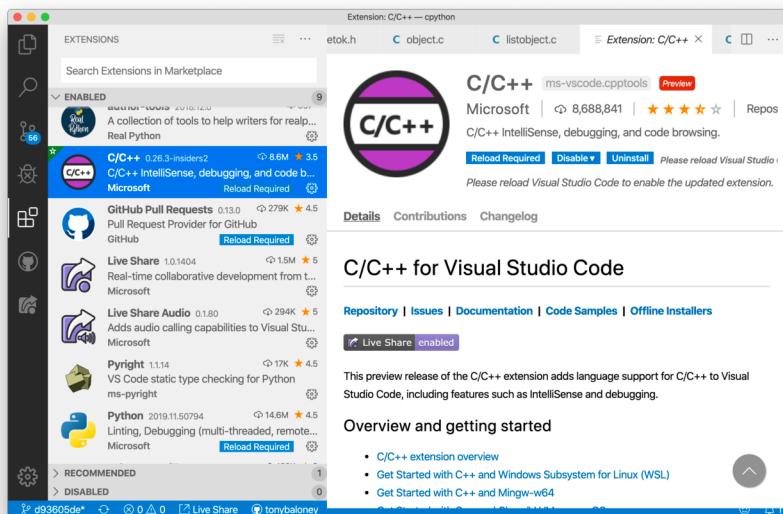
It makes an excellent choice for working with CPython as it supports both C and Python, with an integrated Git interface.

Installing

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

Out-of-the-box, VS Code will have required code editing capabilities but becomes more powerful once you have installed extensions.

The Extensions panel is available by selecting `View > Extensions` from the top menu:



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

Recommended Extensions for This Book

- **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.
- **Restructured Text (`lexstudio.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.

Once you have installed these extensions, you will need to reload the editor.

Because 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` and a terminal will appear below the code editor:

```
78     return 0;
79 }
80
81 static int
82 list_reallocate_exact(PyListObject *self, Py_ssize_t size)
```

Using the Advanced Code Navigation (IntelliSense)

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

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

Include > C listobject.h -> PyList_CheckExact(op)

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

_bisectionmodule.c ~/cython/Modules - References (30)

101 `index = _internal_bisection_right(list, item, lo, hi);`

102 `if (index < 0)`

103 `return NULL;`

104 `if (PyList_CheckExact(list)) {`

105 `if (PyList_Insert(list, index, item) < 0)`

106 `return NULL;`

107 `}`

108 `else {`

109 `result = _PyObject_CallMethodId(list, &PyId_insert,`

110 `if (result == NULL)`

111 `return NULL;`

112 `Py_DECREF(result);`

51

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

 ↳ `_bisectionmodule.c` Modules 2

 ↳ `if (PyList_CheckExact(list)) {`

 ↳ `if (PyList_CheckExact(list)) {`

 ↳ `_elementtree.c` Modules 6

 ↳ `> _textio.c` Modules .io 1

 ↳ `> _pickle.c` Modules 4

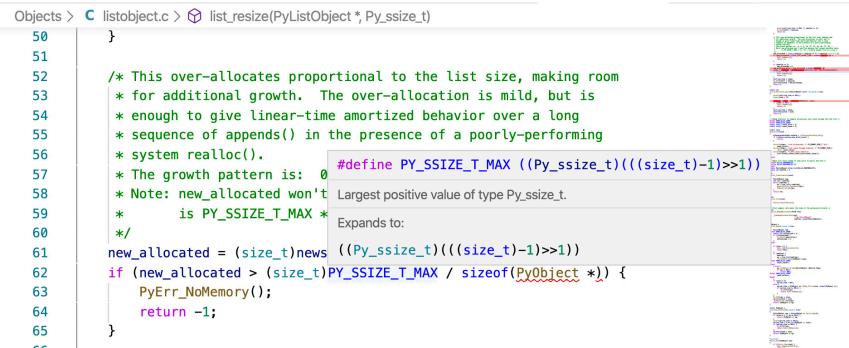
 ↳ `> statement.c` Modules_sqlite 3

 ↳ `> _testcapi module.c` Modules 1

 ↳ `> _cmodule.c` Modules 1

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

By clicking or hovering over a C Macro, the editor will expand that Macro to the compiled code:



```

Objects > C listobject.c > list_resize(PyListObject * Py_ssize_t)
50 } 
51 
52 /* This over-allocates proportional to the list size, making room
53 * for additional growth. The over-allocation is mild, but is
54 * enough to give linear-time amortized behavior over a long
55 * sequence of appends() in the presence of a poorly-performing
56 * system realloc().
57 * The growth pattern is: 0
58 * Note: new_allocated won't
59 *       be PY_SSIZE_T_MAX *
60 */
61 new_allocated = (size_t)news
62 if (new_allocated > (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *))
63     PyErr_NoMemory();
64     return -1;
65 }

```

To jump to the definition of a function, hover over any call to it and press `cmd`+`click` on macOS and `ctrl`+`click` on Linux and Windows.

Configuring the Task and Launch Files

VS Code creates a folder, `.vscode` in the Workspace Directory. Inside this folder, you can create:

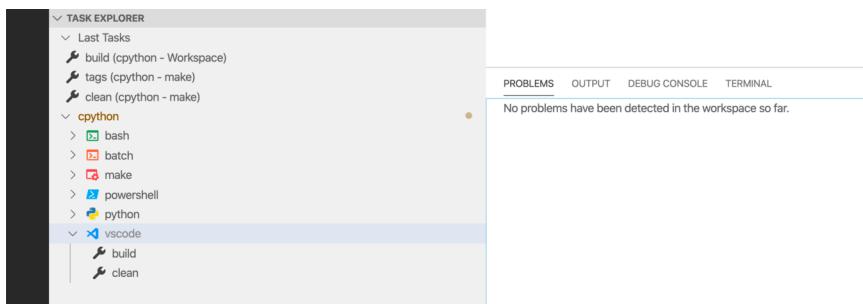
- `tasks.json` for shortcuts to commands that execute your project
- `launch.json` to configure the debugger (see the chapter on Debugging)
- other plugin-specific files

Create a `tasks.json` file inside the `.vscode` directory. If it doesn't exist, create it now. This `tasks.json` 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"  
      }  
    }  
  ]  
}
```

With the `Task Explorer` plugin, you will see a list of your configured tasks inside the **vscode** group:



In the next chapter, you will learn more about the build process for

compiling CPython.

Setting up JetBrains CLion

JetBrains make 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 cannot install C/C++ support into PyCharm, but CLion comes bundled with Python support.

To setup CPython in CLion, install the following plugins:

- [Makefile support](#)

CLion is fully integrated with the CMake system. However, CPython uses **GNU Make**. CLion will give an error saying it cannot locate a `CMakeLists.txt` file when you open the CPython source code directory. There is a workaround to create a `compile_commands.json` file from the compilation steps inside the CPython Makefile.

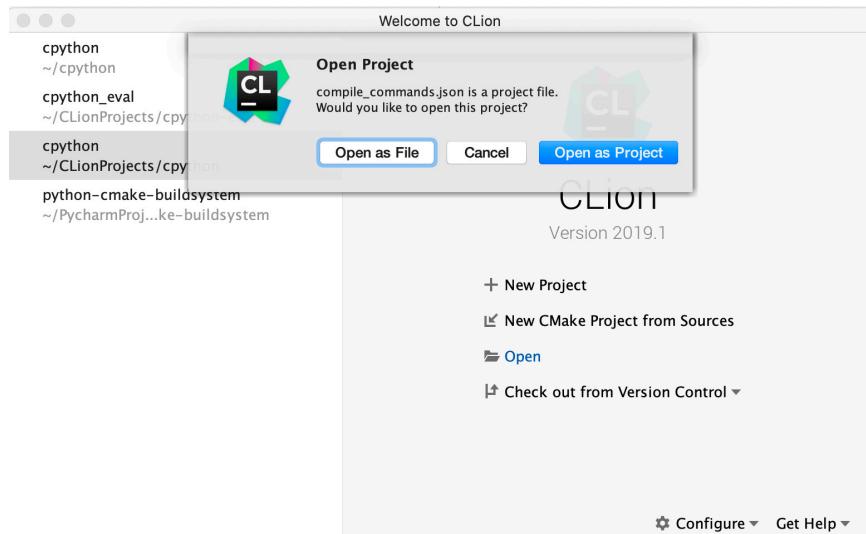
Important

This step assumes you can compile CPython, please read [Compiling CPython for your Operating System](#) and return to this chapter.

We need to create a “Compile Database” using a utility called `compiledb`. At the command line, within the CPython repository:

```
$ pip install compiledb
$ compiledb make
```

Then open CLion and open the `compile_commands.json` file, you will be asked to open file or “Open as Project,” select [Open as Project](#):

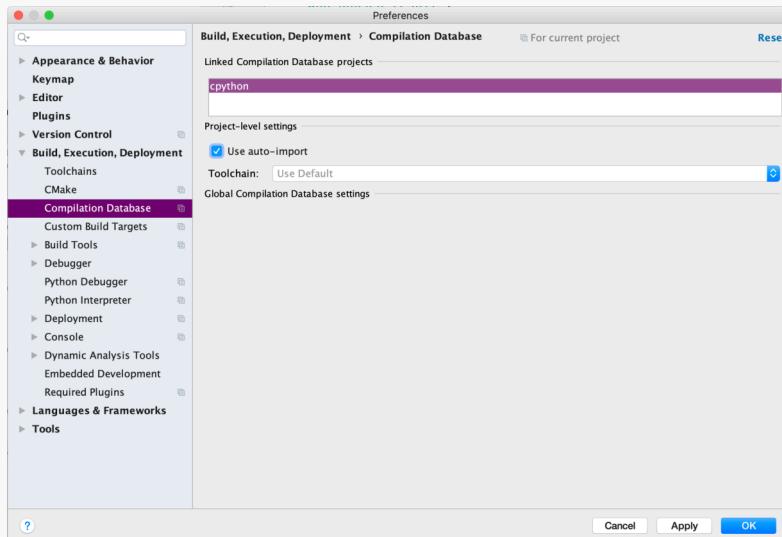


CLion will then open up the CPython source directory as a Project.

Note

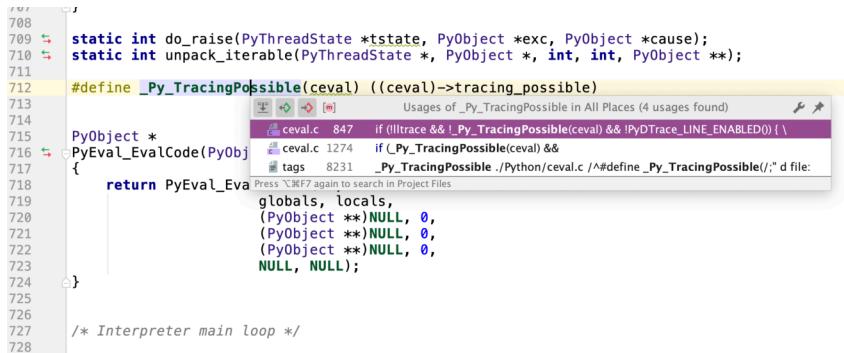
For versions of CLion before 2020.1, you need to link the compile commands and the project together. Go to CLion Settings,

`Build, Execution, Deployment` > `Compilation Database` and check the “Use auto-import” box:



When you open this project in the future, it will have intelligent code navigation based on the compiled version of CPython.

Within the code editor, the shortcut `cmd`+`click` on macOS, and `ctrl`+`click` on Windows and Linux will give in-editor navigation features:



The screenshot shows the CLion code editor with a floating search results window. The code in the editor is:

```

708
709     static int do_raise(PyThreadState *tstate, PyObject *exc, PyObject *cause);
710
711     static int unpack_iterable(PyThreadState *, PyObject *, int, int, PyObject **);
712
713     #define _Py_TracingPossible(ceval) ((ceval)->tracing_possible)
714
715     PyObject *
716     PyEval_EvalCode(PyObject *code, PyObject *globals, PyObject *locals,
717     PyObject **args, PyObject **kwdargs, PyObject **kwargs, PyObject **closure);
718
719     PyObject *
720     PyEval_EvalCodeEx(PyObject *code, PyObject *globals, PyObject *locals,
721     PyObject **args, PyObject **kwdargs, PyObject **kwargs, PyObject **closure,
722     PyObject **closure);
723
724
725     /* Interpreter main loop */
726
727
728

```

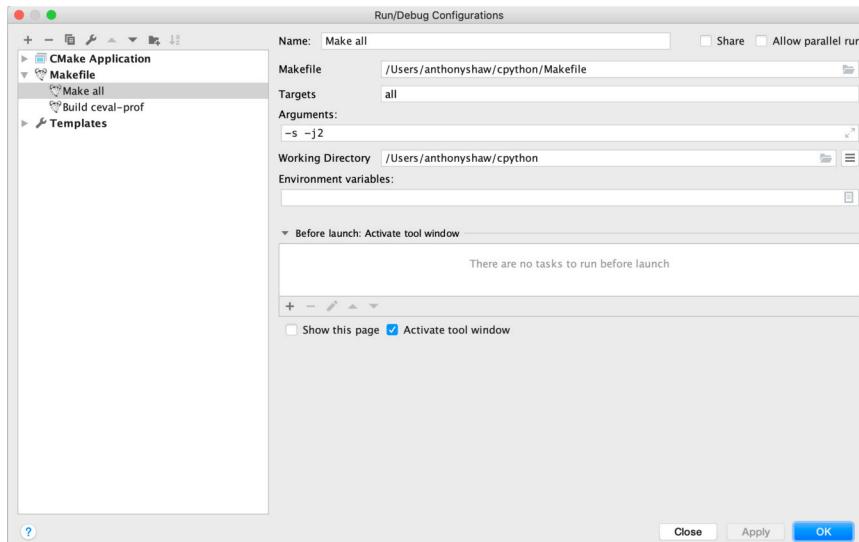
The floating window shows the usage of `_Py_TracingPossible` with 4 usages found. One usage is highlighted in the code editor:

```

713     #define _Py_TracingPossible(ceval) ((ceval)->tracing_possible)
714
715     PyObject *
716     PyEval_EvalCode(PyObject *code, PyObject *globals, PyObject *locals,
717     PyObject **args, PyObject **kwdargs, PyObject **kwargs, PyObject **closure);
718
719     PyObject *
720     PyEval_EvalCodeEx(PyObject *code, PyObject *globals, PyObject *locals,
721     PyObject **args, PyObject **kwdargs, PyObject **kwargs, PyObject **closure,
722     PyObject **closure);
723
724
725     /* Interpreter main loop */
726
727
728

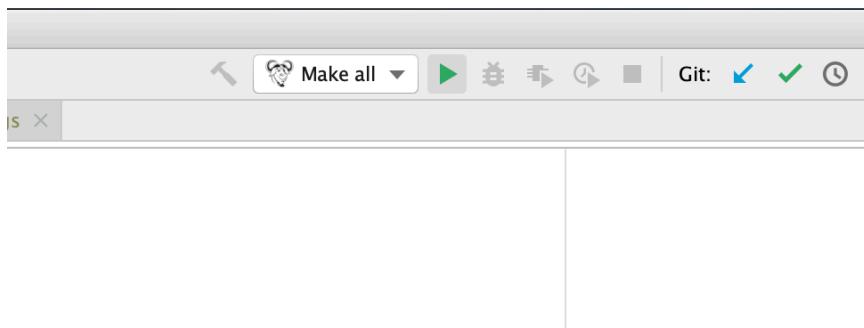
```

You can use the Makefile plugin to create shortcuts for compiling CPython. Select **Run** **Edit Configurations...** to open the “Run/Debug Configurations” window. Inside this window select **[+]** **Makefile** to add a Makefile configuration. Set Make all as the name, all as the target, and `-s j2` as the arguments. Ensure the Working Directory is set to the directory where you have downloaded the source code:



Click **Apply** to add this configuration. You can repeat this step as many times as you like for any of the CPython make targets. See CPython’s Make Targets for a full reference.

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



Now that CLion is set up follow the steps for compiling CPython.

Setting up Vim

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

Note

On most Linux distributions and within the macOS Terminal, `vi` is an alias for `vim`. I'll use the `vim` command, but if you have the alias, `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, Vim can become powerful for both Python and C editing. Vim's extensions are in various locations, like 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.

We will install two plugins:

- [vim-fugitive](#) A status bar for Git with [shortcuts](#) for many Git tasks.
- [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:

```
$ 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 between methods, functions, and macros quickly. Only using text search won't determine a call to a function, or its definition, versus the implementation. An application called [ctags](#) will index source files across a multitude of languages into a plaintext database.

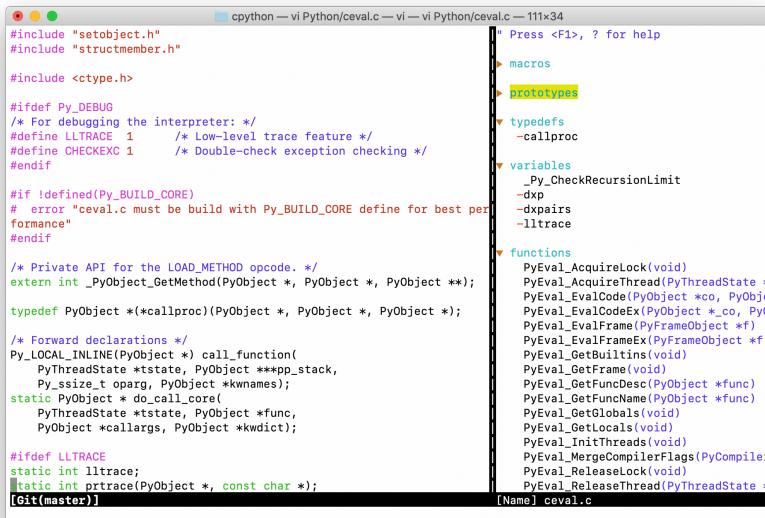
To index CPython's headers for all the C files, and Python files in the standard library, run:

```
$ make tags
```

When you open `vim` now, for example editing the `Python/ceval.c` file:

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

You will see the Git status at the bottom and the functions, macros, and variables on the right-hand pane:



```
#include "setobject.h"
#include "structmember.h"

#include <ctype.h>

#ifndef Py_DEBUG
/* For debugging the interpreter: */
#define LLTRACE 1      /* Low-level trace feature */
#define CHECKEXC 1     /* Double-check exception checking */
#endif

#if !defined(Py_BUILD_CORE)
# error "ceval.c must be build with Py_BUILD_CORE define for best performance"
#endif

/* Private API for the LOAD_METHOD opcode. */
extern int _PyObject_GetMethod(PyObject *, PyObject *, PyObject **);

typedef PyObject *(*callproc)(PyObject *, PyObject *, PyObject *);

/* Forward declarations */
Py_LOCAL_INLINE(PyObject *) call_function(
    PyThreadState *tstate, PyObject ***pp_stack,
    Py_ssize_t oparg, PyObject **kwnames);
static PyObject * do_call_core(
    PyThreadState *tstate, PyObject *func,
    PyObject *callargs, PyObject *kwdict);

#ifndef LLTRACE
static int lltrace;
static int ptrace(PyObject *, const char *);
#endif
```

[Git(master)] [Name] ceval.c

When editing Python files, such as the `Lib/subprocess.py`:

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

The tagbar will show imports, classes, methods, and functions:

```

  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
      self.output = value

  class TimeoutExpired(SubprocessError):
      pass

```

Within `vim` you can switch between tabs as `CTRL+W`, `L` to move to the right-hand pane, using 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`, `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, you don't need to make a decision right here and now. I used multiple environments while writing this book and working on changes to CPython. A critical feature for productivity is debugging, so having a reliable debugger that you can use to explore the runtime and understand bugs will save a lot of time. Especially if you're used to relying on `print()`

functions for debugging in Python, that approach doesn't work in C. You will cover Debugging in full later in this book.

[Leave feedback on this section »](#)

Compiling CPython

Now that you have CPython 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. So 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 work, you require 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.

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

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

Compiling CPython on macOS

Compiling CPython on macOS requires some additional applications and libraries. You will first need the essential C compiler toolkit. “Command Line Development 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 will want to save this app to your Dock, so right-click the Icon and select `Keep in Dock`

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

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

This command will pop up with a prompt to download and install a set of tools, including Git, Make, and the GNU C compiler.

You will also need a working copy of [OpenSSL](#) to use for fetching packages from the PyPi.org website. If you later plan on using this build to install additional packages, SSL validation is required.

The simplest way to install OpenSSL on macOS is by using [Homebrew](#).

Note

If you don't have Homebrew, you can download and install Homebrew 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. Homebrew has a command `brew --prefix [package]` that will give the directory where the 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 if you intend on debugging for development or testing purposes. Debugging CPython is covered extensively in the Debugging chapter.

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

```
$ 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 that you can use to automate the build process.

You can now build the CPython binary by running:

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

See Also

For more help on the options for `make`, see 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, `osaudiodev`, `spwd`, and `_tkinter` would fail to build with this set of instructions. That's okay if you aren't planning on developing against those packages. If you are, then check out the [official dev guide](#) website 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 will

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 3.9.0b1 (tags/v3.9.0b1:97fe9cf, May 19 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 file extension for `.exe`. This extension is *not* because it's a Windows binary! Because macOS has a case-insensitive filesystem and when working with the binary, the developers didn't want people to accidentally refer to the directory `Python/` so `.exe` was appended to avoid ambiguity. If you later run `make install` or `make altinstall`, it will rename the file back to `python` before installing it into your system.

Compiling CPython on Linux

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

For Fedora Core, RHEL, CentOS, or other yum-based systems:

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

For Debian, Ubuntu, or other apt-based systems:

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

Then install some additional required packages.

For Fedora Core, RHEL, CentOS or other yum-based systems:

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

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 `--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 section [A Quick Primer on Make](#).

Review the output to ensure that there weren't issues compiling the `_ssl` module. If there were, 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 3.9.0b1 (tags/v3.9.0b1:97fe9cf, May 19 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

From your source repository, if you're happy with your changes and want to use them inside your system, you can install it as a custom version.

For macOS and Linux, you can use the `altinstall` command, which won't create symlinks for `python3` and install 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 which 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 haven't spent much time with it. For C, C++, and other compiled languages, the number of commands you need to execute to load, link, and compile your code in the right order can be exhaustive. 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 copy+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 `Makefile`.

The generated `Makefile` is similar to a shell script, 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 only runs two commands.

The convention for executing any make target is:

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

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

Make has many options, so here are the ones I think you'll find useful throughout this book:

Option	Use
<code>-d, --debug[=FLAGS]</code>	Print various types of debugging information
<code>-e, --environment-overrides</code>	Environment variables override makefiles
<code>-i, --ignore-errors</code>	Ignore errors from commands
<code>-j [N], --jobs[=N]</code>	Allow N jobs at once (infinite jobs otherwise)
<code>-k, --keep-going</code>	Keep going when some targets can't be made
<code>-l [N], --load-average[=N]</code>	Only start multiple jobs if load < N
<code>--max-load[=N]</code>	
<code>-n, --dry-run</code>	Print commands instead of running them.
<code>-s, --silent</code>	Don't echo commands.
<code>-S, --stop</code>	Turns off <code>-k</code> .

In the next section and throughout the book, I'm going to ask you to run `make` with the options:

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

The `-j2` flag allows `make` to run 2 jobs simultaneously. If you have 4

or more cores, you can change this to 4 or larger 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 is happening, remove the `-s` flag.

CPython’s Make Targets

For both Linux and macOS, you will find yourself needing to clean up files, build, or to refresh configuration.

There are a number of useful make targets built into CPython’s Makefile:

Build Targets

Target	Purpose
<code>a11</code> (default)	Build the compiler, libraries and modules
<code>profile-opt</code>	Compile the Python binary with profile guided optimization
<code>clinic</code>	Run “Argument Clinic” over all source files)
<code>sharedmods</code>	Build the shared modules
<code>regen-all</code>	Regenerate all generated files

Test Targets

Target	Purpose
<code>test</code>	Run a basic set of regression tests
<code>testall</code>	Run the full test suite twice - once without <code>.pyc</code> files, and once with
<code>quicktest</code>	Run a faster set of regression tests, excluding the tests that take a long time
<code>testuniversal</code>	Run the test suite for both architectures in a Universal build on OSX
<code>coverage</code>	Compile and run tests with <code>gcov</code>
<code>coverage-lcov</code>	Create coverage HTML reports

Cleaning Targets

The primary clean 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, try `clobber`. For completely cleaning out an environment before distribution, run the `distclean` target.

Target	Purpose
<code>check-clean-src</code>	Check that the source is clean when building out of source
<code>cleantest</code>	Remove "test_python_%" directories of previous failed test jobs
<code>clean</code>	Remove <code>pyc</code> files, compiled libraries and profiles
<code>pycremoval</code>	Remove <code>pyc</code> files
<code>docclean</code>	Remove built documentation in <code>Doc/</code>
<code>profile-removal</code>	Remove any optimization profiles
<code>clobber</code>	Same as <code>clean</code> , but also removes libraries, tags, configurations and builds
<code>distclean</code>	Same as <code>clobber</code> , but also removes anything generated from source, e.g. <code>Makefile</code>

Installation Targets

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

After installing using `make install`, the command `python3` will now link to your compiled binary.

Whereas, using `make altinstall`, only `python$(VERSION)` will be installed and the existing link for `python3` will remain intact.

Target	Purpose
<code>install</code>	Installs shared libraries, binaries and documentation. Will run <code>commoninstall</code> , <code>bininstall</code> and <code>maninstall</code>
<code>bininstall</code>	Installs all the binaries, e.g. <code>python</code> , <code>idle</code> , <code>2to3</code>
<code>altinstall</code>	Installs shared libraries, binaries and documentation with the version suffix

Target	Purpose
<code>maninstall</code>	Install the manuals
<code>altmaninstall</code>	Install the versioned manuals
<code>altdbininstall</code>	Install the <code>python</code> interpreter, with the version affixed, e.g. <code>python3.9</code>
<code>commoninstall</code>	Install shared libraries and modules
<code>libinstall</code>	Install shared libraries
<code>sharedinstall</code>	Dynamically loaded modules

Miscellaneous Targets

Target	Purpose
<code>python-config</code>	Generate the <code>python-config</code> script
<code>recheck</code>	Rerun configure with the same options as it was run last time
<code>autoconf</code>	Regenerate configure and <code>pyconfig.h.in</code>
<code>tags</code>	Create a tags file for vi
<code>TAGS</code>	Create a tags file for emacs
<code>smelly</code>	Check that exported symbols start <code>Py</code> or <code>_Py</code> (see PEP7)

Compiling CPython on Windows

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

The first is to compile from the command line, this still requires the Microsoft Visual C++ compiler, which comes with Visual Studio. The second is to open the `PCBuild\pcbuild.sln` from Visual Studio and build directly.

Installing the Dependencies

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

Inside the `PCBuild` folder there is a `.bat` file that automates this for you.

Open up a command-line prompt 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 the command line or Visual Studio.

Compiling From the Command Line

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

If you do any debugging, 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 line:

```
> amd64\python_d.exe
```

```
Python 3.9.0b1 (tags/v3.9.0b1:97fe9cf, May 19 2020, 10:00:00)
```

```
[MSC v.1922 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You are now inside the REPL of your compiled CPython binary.

To compile a release binary:

```
> 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](https://www.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
<code>-p</code>	Build platform CPU architecture	<code>x64, Win32</code> (default), <code>ARM, ARM64</code>
<code>-c</code>	Build configuration	<code>Release</code> (default), <code>Debug, PGInstrument</code> OR <code>PGUpdate</code>
<code>-t</code>	Build target	<code>Build</code> (default), <code>Rebuild, Clean, CleanAll</code>

Flags

Here are some optional flags you can use for `build.bat`. For a full list, run `build.bat -h`.

Flag	Purpose
<code>-v</code>	Verbose mode. Show informational messages during build
<code>-vv</code>	Very verbose mode. Show detailed messages during build
<code>-q</code>	Quiet mode. Only show warning and errors during build
<code>-e</code>	Download and install external dependencies (default)
<code>-E</code>	Don't download and install external dependencies
<code>--pgo</code>	Build with profile-guided-optimization
<code>--regen</code>	Regenerate all grammar and tokens, used when you update the language

Compiling From Visual Studio

Inside the `PCBuild` folder is a Visual Studio project file, `PCBuild\pcbuild.sln`, for building and exploring CPython source code.

When the Solution is loaded, it will prompt you to retarget the project's inside the Solution to the version of the C/C++ compiler you have installed. Visual Studio will also target the release of the Windows SDK you have installed.

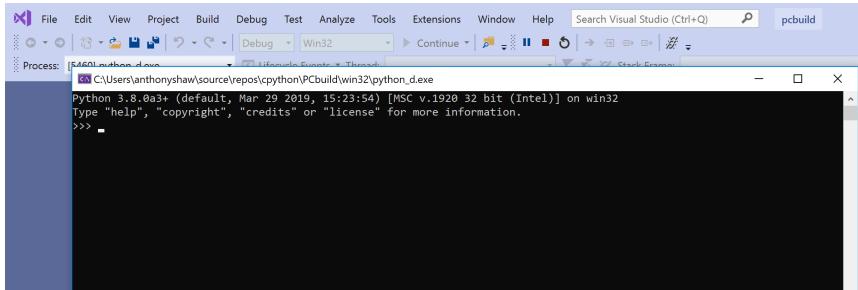
Ensure that you change the Windows SDK version to the newest installed version and the platform toolset to the latest version. If you missed this window, you can right-click on the Solution in the *Solutions and Projects* window and click *Retarget Solution*.

Navigate to `Build > Configuration Manager` and ensure the “Active solution configuration” is set to `Debug`, and the “Active Solution Platform” is set to `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 *Windows Kits* inside your Start Menu, and *Windows Software Development Kit* inside of that menu.

The build stage could take 10 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 start in Debug mode straight into the REPL:

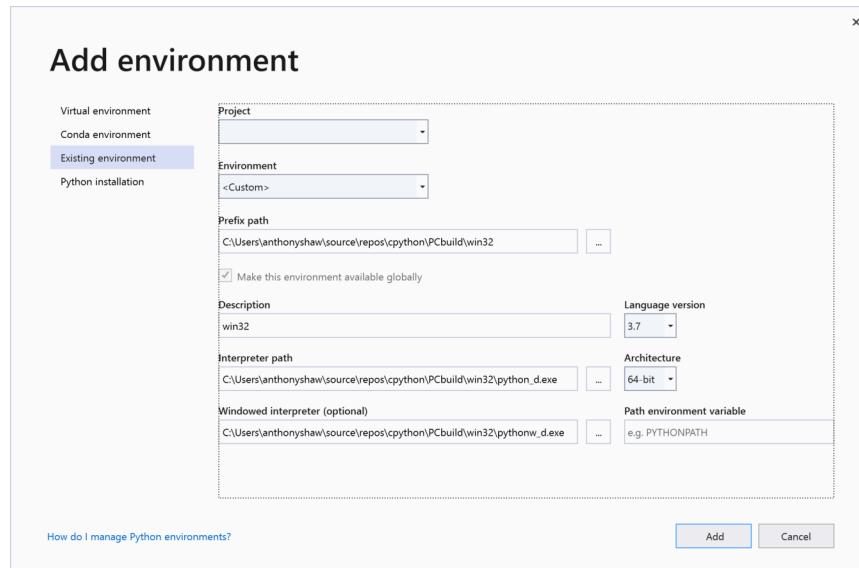


You can run the Release build by changing the build configuration from *Debug* to *Release* on the top menu bar and rerunning **Build** **Build Solution**. You now have both Debug and Release versions of the CPython binary within **PCBuild** **amd64**.

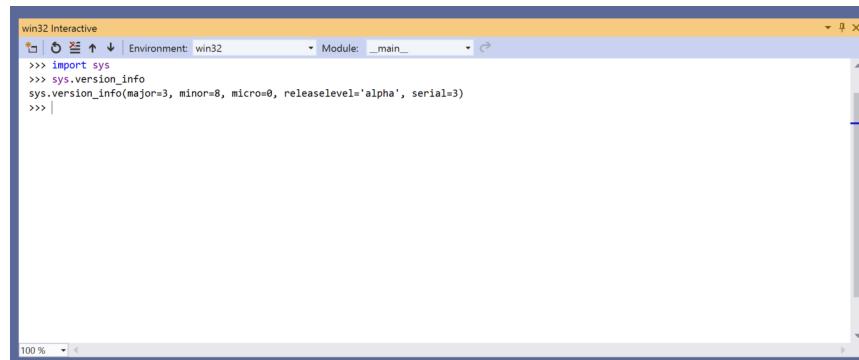
You can set up Visual Studio to be able to open a REPL with either the Release or Debug build by choosing **Tools** **Python** **Python Environments** from the top menu:

In the Python Environments panel, click *Add Environment* and then target the Debug or Release binary. The Debug binary will end in `_d.exe`. For example, `python_d.exe` and `pythonw_d.exe`. You will most likely want to use the debug binary as it comes with Debugging support in Visual Studio and will be useful for 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:

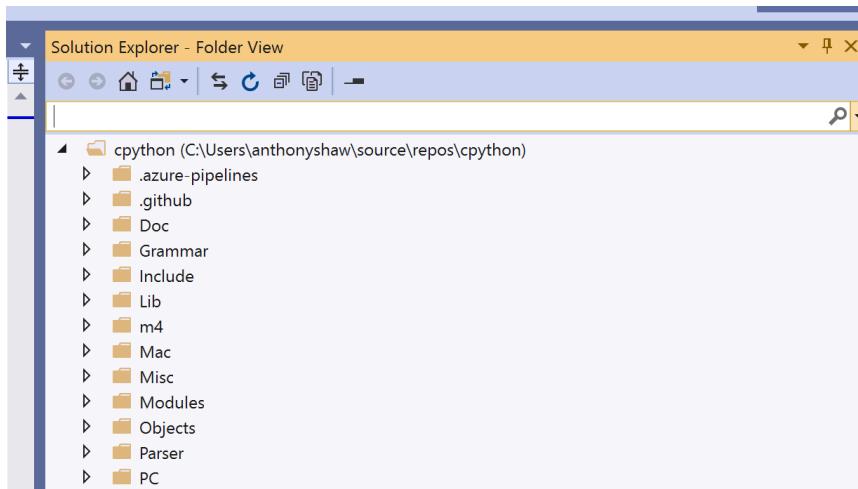


Start a REPL session by clicking [Open Interactive Window](#) in the Python Environments window and you will see the REPL for the compiled version of Python:



During 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 on the toggle button next to the Home icon to switch to Folder view:



Profile Guided Optimization

The macOS, Linux, and Windows build processes have flags for “PGO,” or “Profile Guided Optimization.” PGO is not 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 created is then analyzed, and the compiler will make changes to the binary that would 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 throughout this book I've excluded it from the list of recommended steps to keep your compilation time short. If you want to distribute a custom compiled version of CPython into a production environment, 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 cannot 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 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” to reduce the stack-size.
- **Virtual Call Speculation and Inlining** - If a virtual function call frequently targets a certain function, 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 set of pages (locality). It minimizes the number of pages used, which minimizes memory overhead.
- **Hot-Spot Optimization** - Functions where the program spends the most execution time can be optimized for speed.
- **Function Layout Optimization** - After analyzing 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 10 cases in a `switch` statement and one is used 95% of the time, then it 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 can use this knowledge throughout the book as you explore and adapt the source code.

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

[Leave feedback on this section »](#)

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 speak in Japanese.

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

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

A 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 systems 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 does compile your code when it runs for the first time. This step is invisible to the regular user.

Python code is not compiled into machine code; it is 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, 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, implying 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 located in 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, 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 (topics you will cover in this chapter). Popular compiler-compilers include GNU Bison, Yacc, and ANTLR.

See Also

If you want to learn more about parsers, 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 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 filesystem](#), or [interacting with the display](#) are all written in C.

It made sense for Python's extensibility layer to be focused on the C language. Later in this book, you will 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 of 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 of the grammatical structure for the language before it tries to execute it.

Note

For the rest of this tutorial, `./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 machine-readable format. Inside the documentation is a detailed explanation of the Python language. What is allowed and how each statement should behave.

Language Documentation

Located inside the `Doc` \rightarrow `reference` directory are [reStructuredText](#) explanations of each 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	
└── <i>compound_stmts.rst</i>	Compound statements like if, while, for and function definitions
└── <i>datamodel.rst</i>	Objects, values and types
└── <i>executionmodel.rst</i>	The structure of Python programs
└── <i>expressions.rst</i>	The elements of Python expressions
└── <i>grammar.rst</i>	Python's core grammar (referencing Grammar/Grammar)
└── <i>import.rst</i>	The import system
└── <i>index.rst</i>	Index for the language reference
└── <i>introduction.rst</i>	Introduction to the reference documentation
└── <i>lexical_analysis.rst</i>	Lexical structure like lines, indentation, tokens and keywords
└── <i>simple_stmts.rst</i>	Simple statements like assert, import, return and yield
└── <i>toplevel_components.rst</i>	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, and the machine-readable specification is housed in a single file, `Grammar` ▶ `Grammar`.

The Grammar File

Important

This section refers to the grammar file used by the “old parser”. At the time of publishing, the “new parser” (the PEG parser) is experimental and unfinished.

For releases of CPython up to and including 3.8, the pgen parser is the default. For releases of CPython 3.9 and above, the PEG parser is the default. The old parser can be enabled with `-x oldparser` on the command line.

The Tokens file is used by both parsers.

The Grammar file is written in a context-notation called [Backus-Naur Form \(BNF\)](#). BNF is not specific to Python and is often used as the notation for grammar in many other languages.

The concept of grammatical structure in a programming language is inspired by [Noam Chomsky’s work on Syntactic Structures](#) in the 1950s!

Python’s grammar file uses the Extended-BNF (EBNF) specification with regular-expression syntax. So, in the grammar file you can use:

- `*` 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:

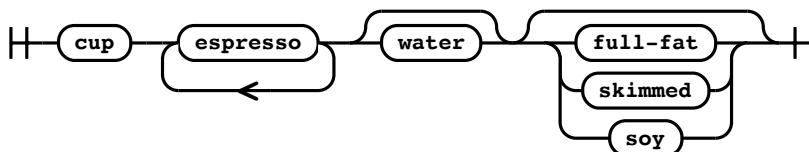
- It must have a cup
- It must include at least one shot of espresso and can contain multiple

- It can have milk, but it is optional
- There are many types of milk you can put into coffee, like full fat, skimmed and soy

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

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

In this chapter, grammar is visualized with railroad diagrams. This diagram is the 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.

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

```
with_stmt: 'with' with_item (',' with_item)* ':' suite
with_item: test ['as' expr]
```

Anything in quotes is a string literal, known as a terminal. Terminals are how keywords are recognized. The `with_stmt` is specified as:

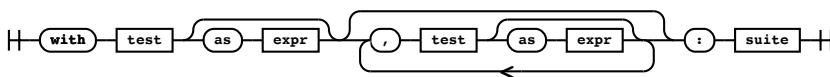
1. Starting with the word `with`
2. Followed by a `with_item`, which is a `test` and (optionally), the word `as`, and an expression
3. Following one or many `with_item`, each separated by a comma

4. Ending with a :
5. Followed by a suite

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

- **suite** refers to a block of code with one or multiple statements
- **test** refers to a simple statement that is evaluated
- **expr** refers to a simple expression

Visualized in a Railroad Diagram, the `with` statement looks like this:



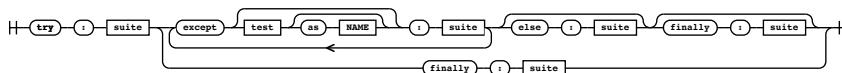
As a more complex example, the `try` statement is defined as:

```
try_stmt: ('try' ':' suite
          ((except_clause ':' suite)+
           ['else' ':' suite]
           ['finally' ':' suite] |
           'finally' ':' suite))
except_clause: 'except' [test ['as' NAME]]
```

There are two uses of the `try` statement:

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

Or, 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, the grammar is defined in [Grammar ▶ Grammar](#).

Using the Parser Generator

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

Parser Tables are a list of potential parser states. When parse trees become complex, they ensure that grammar cannot be ambiguous.

The Parser Generator

A parser generator works by converting the EBNF statements into a Non-deterministic Finite Automaton (NFA). The NFA states and transitions are resolved and consolidated into a Deterministic Finite Automaton (DFA).

The DFAs are used by the parser as parsing tables. This technique was [formed at Stanford University](#) and developed in the 1980s, just before the advent of Python. CPython's parser generator, `pgen`, is unique to the CPython project.

The `pgen` application is was rewritten in Python 3.8 from C to Python as [Parser ▶ pgen ▶ pgen.py](#).

It is executed as :

```
$ ./python -m Parser.pgen [grammar] [tokens] [graminit.h] [graminit.c]
```

It is normally executed from the build scripts, not directly.

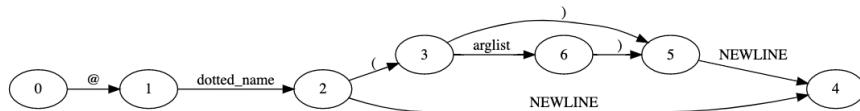
The DFA and NFA don't have a visual output, but [there is a branch of CPython](#) with a directed graph output. `decorator grammar` is defined

in Grammar/Grammar as:

```
decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
```

The parser generator creates a complex NFA graph of 11 states. Each of the states is numerically represented (with hints on their name in the grammar). The transitions are referred to as ‘arcs.’

The DFA is simpler than the NFA, with the paths reduced:



The NFA and DFA graphs are only useful for debugging the design of complex grammars.

We will use Railroad Diagrams for representing grammar instead of DFA or NFA graphs.

As an example, this diagram represents the paths that can be taken for the decorator statement:



Regenerating Grammar

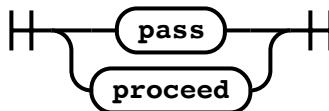
To see `pgen` in action, let’s change part of the Python grammar. Search `Grammar` ▶ `Grammar` for `pass_stmt` to see the definition of a `pass` statement:

```
pass_stmt: 'pass'
```



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

```
pass_stmt: 'pass' | 'proceed'
```



Next, rebuild the grammar files by running `pgen`. CPython comes with scripts to automate `pgen`.

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

```
$ make regen-grammar
```

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

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

You should see an output showing that the new `Include\graminit.h` and `Python\graminit.c` files have been regenerated.

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

If the code compiled successfully, 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 -X oldparser

Python 3.9.0b1 (tags/v3.9.0b1:97fe9cf, May 19 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, we'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 are used to make it simpler to refer to 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 `pgen`.

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

Note

There are two tokenizers in the CPython source code. One tokenizer written in Python demonstrated here, and another written in C. The tokenizer written in Python is a utility, and the Python interpreter uses the one written in C. They have identical output and behavior. The version written in C is designed for performance, and the module in Python is designed for debugging.

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

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

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

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

```
$ ./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/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 blank line at the end
- A `DEDENT` to close the function declaration
- An `ENDMARKER` to end the file

It is best practice to have a blank line at the end of your Python source files. If you omit it, CPython adds it 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 tokenizer, you can run Python with the `-d` flag. Using the `test_tokens.py` script you created earlier, run it with the following:

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

Token NAME/'def' ... It's a keyword
DFA 'file_input', state 0: Push 'stmt'
DFA 'stmt', state 0: Push 'compound_stmt'
...
Token NEWLINE/'' ... It's a token we know
```

```
DFA 'funcdef', state 5: [switch func_body_suite to suite] Push 'suite'
DFA 'suite', state 0: Shift.
Token INDENT/'' ... It's a token we know
DFA 'suite', state 1: Shift.
Token NAME/'proceed' ... It's a keyword
DFA 'suite', state 3: Push 'stmt'
...
ACCEPT.
```

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

Note

To clean up your code, revert the change in `Grammar`▶`Grammar`, regenerate the grammar again, then clean the build, and recompile:

For macOS or Linux:

```
$ git checkout -- Grammar/Grammar
$ make regen-grammar
$ make clobber
$ make -j2 -s
```

Or for Windows:

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

A More Complex Example

Adding `proceed` as an alternate keyword for `pass` is a simple change, the parser generator does the work of matching '`proceed`' as a literal for the `pass_stmt` token. This new keyword works without any changes to

the compiler.

In practice, most changes to the grammar are more complicated.

Python 3.8 introduced assignment expressions, with the format `:=`. An assignment expression both assigns a value to a name and returns the value of the named variable.

One of the statements impacted by the addition of assignment expressions to the Python language was the `if` statement.

Prior to 3.8, the `if` statement was defined as:

- The keyword `if` followed by a test, then a `:`
- A nested series of statements (`suite`)
- Zero-or-more `elif` statements, which are followed by a test, a `:` and a `suite`
- An optional `else` statement, which is followed by a `:` and a `suite`

In the grammar this was represented as:

```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

Visualized, this looks like:



To support assignment expressions, the change needed to be backward compatible. The use of `:=` in an `if` statement, therefore, had to be optional.

The `test` token type used in the `if` statement is generic between many statements. For example, the `assert` statement is followed by a `test` (and then optionally a second `test`).

```
assert_stmt: 'assert' test [',' test]
```

An alternate test token type was added in 3.8, so that the grammar could be prescriptive about which statements should support assignment expressions and which should not.

This is called `namedexpr_test` and is defined in the Grammar as:

```
namedexpr_test: test [':=' test]
```

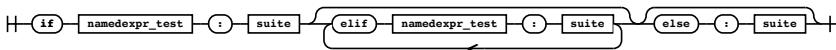
Or, visualized in a Railroad Diagram:



The new syntax for the if statement was changed to replace `test` with `namedexpr_test`:

```
if_stmt: 'if' namedexpr_test ':' suite ('elif' namedexpr_test ':' suite)
       ['else' ':' suite]
```

Visualized in a Railroad Diagram:



To distinguish between `:=` and the existing COLON (`:`) and EQUAL (`=`) token, the following token was then added to `Grammar` ▶ `Tokens`

```
COLONEQUAL
```

```
' :='
```

This is not the only change required to support assignment expressions. The change altered many parts of the CPython compiler, as seen in [the Pull Request](#).

See Also

For more information on CPython’s parser generator, the author of `pgen` has recorded a presentation on the implementation and design: “[The soul of the beast](#)” at PyCon Europe 2019.

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-equals” operator.

In practice, changes to the Python grammar have to be carefully considered and discussed. There are two reasons for the level of scrutiny:

1. Having “too many” language features, or a complex grammar would change the ethos of Python being a simple and readable language
2. Changes to grammar introduce backward-incompatibilities, which create work for all developers

If a Python Core Developer proposes a change to the grammar, it must be proposed as a Python Enhancement Proposal (PEP). All PEPs are numbered and indexed on the PEP index. [PEP 5](#) documents the guidelines for evolving the language and specifies that changes must be proposed in PEPs.

Members can also suggest changes to the language outside of the core development group through the [python-ideas mailing list](#).

You can see the drafted, rejected, and accepted PEPs for future versions of CPython in the [PEP index](#). Once a PEP has consensus, and the draft has been finalized, the Steering Council must accept or reject it. The mandate of the Steering Council, defined in [PEP 13](#), states that they shall work to “Maintain the quality and stability of the Python language and CPython interpreter.”

[Leave feedback on this section »](#)

Configuration and Input

Now that you've seen the Python grammar, its time to explore how code gets input into a state that can be executed.

There are many ways Python code can be run in CPython. Here are some of the most commonly used:

1. By running `python -c` and a Python string
2. By running `python -m` and the name of a module
3. By running `python [file]` with the path to a file that contains Python code
4. By piping Python code into the `python` executable over `stdin`, e.g.,
`cat [file] | python`
5. By starting a REPL and executing commands one at a time
6. By using the C API and using Python as an embedded environment

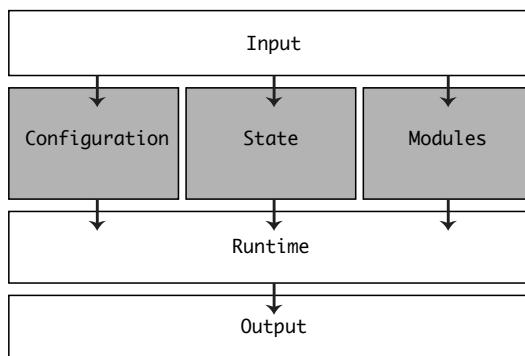
See Also

Python has so many ways to execute scripts; it can be a little overwhelming. Darren Jones has put together a [great course at realpython.com on running Python scripts](#) if you want to learn more.

To execute any Python code, the interpreter needs:

- A module to execute
- A state to hold information such as variables
- Configuration, such as which options are enabled

With these three components, the interpreter can execute code and provide an output:



Note

Similar to the [PEP8](#) style guide for Python code, there is [PEP7](#) for the CPython C code.

There are some naming standards for C source code:

- Use a `Py` prefix for public functions, never for static functions. The `Py_` prefix is reserved for global service routines like `Py_FatalError`. Specific groups of routines (like specific object type APIs) use a longer prefix, such as `PyString_` for string functions.
- Public functions and variables use MixedCase with underscores, like this: `PyObject_GetAttr()`, `Py_BuildValue()`, `PyExc_TypeError()`.
- Occasionally an “internal” function has to be visible to the loader. Use the `_Py` prefix for this, for example, `_PyObject_Dump()`.
- Macros should have a MixedCase prefix and then use upper case, for example `PyString_AS_STRING`, `Py_PRINT_RAW`.

Unlike PEP8, there are few tools for checking the compliance of PEP7. This task is instead done by the core developers as part of code reviews. As with any human-operated process, it isn’t perfect so you will likely find code that does not adhere to PEP7.

The only bundled tool is a script called `smelly.py`, which you can execute using the `make smelly` target on Linux or macOS, or via the command line:

```
$ ./python Tools/scripts/smelly.py
```

This will raise an error for any symbols that are in `libpython` (the shared CPython library) that do not start `Py`, or `_Py`.

Configuration State

Before any Python code is executed, the CPython runtime first establishes the configuration of the runtime and any user-provided options.

The configuration of the runtime is in three data structures, defined in [PEP587](#):

1. `PyPreConfig`, used for pre-initialization configuration
2. `PyConfig`, used for the runtime configuration
3. The compiled configuration of the CPython interpreter

Both data structures are defined in `Include` \rightarrow `cpython` \rightarrow `initconfig.h`.

Pre-Initialization Configuration

The pre-initialization configuration is separate to the runtime configuration as it's properties relate to the Operating System or user environment.

The three primary functions of the `PyPreConfig` are:

- Setting the Python memory allocator
- Configuring the `LC_CTYPE` locale
- Setting the UTF-8 mode ([PEP540](#))

The `PyPreConfig` type contains the following fields; all of type `int`:

Name	Purpose
<code>allocator</code>	Name of the memory allocator (e.g. <code>PYMEM_ALLOCATOR_MALLOC</code>). Run <code>./configure --help</code> for more information on the memory allocator
<code>configure_locale</code>	Set the <code>LC_CTYPE</code> locale to the user preferred locale. If equal to 0, set <code>coerce_c_locale</code> and <code>coerce_c_locale_warn</code> to 0
<code>coerce_c_locale</code>	If equal to 2, coerce the C locale; if equal to 1, read the <code>LC_CTYPE</code> locale to decide if it should be coerced
<code>coerce_c_locale_warn</code>	If non-zero, emit a warning if the C locale is coerced
<code>dev_mode</code>	See <code>PyConfig.dev_mode</code>

Name	Purpose
isolated	Enable isolated mode: sys.path contains neither the script's directory nor the user's site-packages directory
legacy_windows_fs_encoding	(_Windows only_) If non-zero, disable UTF-8 Mode, set the Python filesystem encoding to mbcs
parse_argv	If non-zero, Py_PreInitializeFromArgs() and Py_PreInitializeFromBytesArgs() parse from command line arguments
use_environment	See PyConfig.use_environment
utf8_mode	If non-zero, enable the UTF-8 mode

Related Source Files

The source files relating to `PyPreConfig` are:

File	Purpose
<code>Python>initconfig.c</code>	Loads the configuration from the system environment and merges it with any command line flags
<code>Include>cpython>initconfig.h</code>	Defines the initialization configuration data structure

Runtime Configuration Data Structure

The second stage configuration is the runtime configuration. The runtime configuration data structure in `PyConfig` includes values, such as:

- Runtime flags for modes like debug and optimized
- The mode of execution, e.g. script file, stdin or a module
- Extended options, specified by `-X <option>`
- Environment variables for runtime settings

The configuration data is used by the CPython runtime to enable and disable features.

Setting Runtime Configuration with the Command Line

Python also comes with several [Command Line Interface Options](#).

As an example, CPython has a mode called **verbose mode**. This is primarily aimed at developers for debugging CPython.

In Python you can enable verbose mode with the `-v` flag. In verbose mode, Python will print messages to the screen when modules are loaded:

```
$ ./python -v -c "print('hello world')"

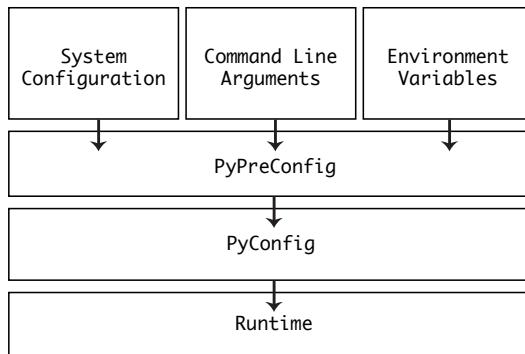
# installing zipimport hook
import zipimport # builtin
# installed zipimport hook
...
```

You will see a hundred lines or more with all the imports of your user site-packages and anything else in the system environment.

Because runtime configuration can be set in several ways, configuration settings have levels of precedence over each other. The order of precedence for verbose mode is:

1. The default value for `config->verbose` is hardcoded to `-1` in the source code.
2. The environment variable `PYTHONVERBOSE` is used to set the value of `config->verbose`.
3. If the environment variable does not exist, then the default value of `-1` will remain.
4. In `config_parse_cmdline()` within `Python > initconfig.c`, the command line flag is used to set the value, if provided.
5. This value is copied to a global variable, `Py_VerboseFlag` by `_Py_GetGlobalVariablesAsDict()`.

All `PyConfig` values follow the same sequence and order of precedence:



Viewing Runtime Flags

CPython interpreters have a set of runtime flags. These flags are advanced features used for toggling CPython specific behaviors. Within a Python session, you can access the runtime flags, like verbose mode and quiet mode, by using the `sys.flags` named tuple. All `-x` flags are available inside the `sys._xoptions` dictionary:

```

$ ./python -X dev -q

>>> import sys
>>> sys.flags
sys.flags(debug=0, inspect=0, interactive=0, optimize=0,
dont_write_bytecode=0, no_user_site=0, no_site=0,
ignore_environment=0, verbose=0, bytes_warning=0,
quiet=1, hash_randomization=1, isolated=0,
dev_mode=True, utf8_mode=0)

>>> sys._xoptions
{'dev': True}
  
```

Build Configuration

As well as the runtime configuration in `Python` → `cpython` → `initconfig.h`, there is also a build configuration. This is located inside `pyconfig.h` in

the root folder. This file is created dynamically in the `./configure` step in the build process for macOS/Linux, or by `build.bat` in Windows.

You can see the build configuration by running:

```
$ ./python -m sysconfig

Platform: "macosx-10.15-x86_64"
Python version: "3.9"
Current installation scheme: "posix_prefix"

Paths:
  data = "/usr/local"
  include = "/Users/anthonyshaw/CLionProjects/cpython/Include"
  platinclude = "/Users/anthonyshaw/CLionProjects/cpython"
...
```

Build configuration properties are compile-time values used to select additional modules to be linked into the binary. For example, debuggers, instrumentation libraries, and memory allocators are all set at compile time.

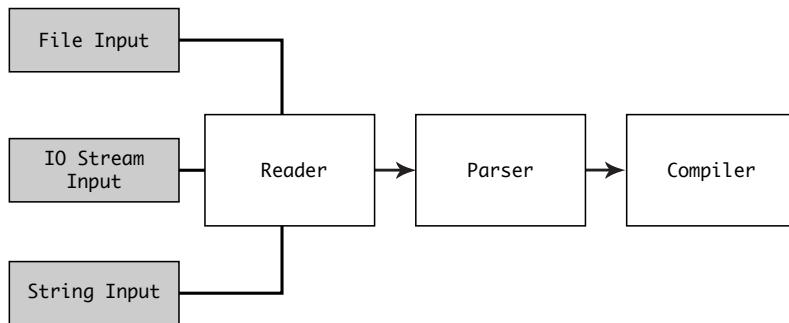
With the three configuration stages, the CPython interpreter can now take input and process text into executable code.

Building a Module From Input

Before any code can be executed, it must be compiled into a module from an input. As discussed before, inputs can vary in type:

- Local files and packages
- I/O Streams, e.g., `stdin` or a memory pipe
- Strings

Inputs are read and then passed to the parser, and then the compiler.



Due to this flexibility, a large portion of the CPython source code is dedicated to processing inputs to the CPython parser.

Related Source Files

There are two main files dealing with the command line interface:

File	Purpose
Lib ➤ <code>runpy.py</code>	Standard Library module for importing Python modules and executing them
Modules ➤ <code>main.c</code>	Functions wrapping the execution of external code, e.g. from a file, module, or input stream
Programs ➤ <code>python.c</code>	The entry point for the <code>python</code> executable for Windows, Linux and macOS. Only serves as a wrapper for <code>Modules/main.c</code> .
Python ➤ <code>pythonrun.c</code>	Functions wrapping the internal C APIs for processing inputs from the command line

Reading Files/Input

Once CPython has the runtime configuration and the command line arguments, it can load the code it needs to execute.

This task is handled by the `pymain_main()` function inside `Modules ➤ main.c`.

Depending on the newly created `PyConfig` instance, CPython will now

execute code provided via several options.

Input String From the Command Line

C_{Py}thon can execute a small Python application at the command line with the `-c` option. For example to execute `print(2 ** 2)`:

```
$ ./python -c "print(2 ** 2)"
```

```
4
```

The `pymain_run_command()` function is executed inside `Modules` \rightarrow `main.c` taking the command passed in `-c` as an argument in the C type `wchar_t*`.

Note

The `wchar_t*` type is often used as a low-level storage type for Unicode data across C_{Py}thon as the size of the type can store UTF8 characters.

When converting the `wchar_t*` to a Python string, the `Objects` \rightarrow `unicodeobject.c` file has a helper function `PyUnicode_FromWideChar()` that returns Unicode string. The encoding to UTF8 is then done by `PyUnicode_AsUTF8String()`.

Python Unicode Strings are covered in depth in the Unicode String Type section of the Objects and Types chapter.

Once this is complete, `pymain_run_command()` will then pass the Python bytes object to `PyRun_SimpleStringFlags()` for execution.

The `PyRun_SimpleStringFlags()` function is part of `Python` \rightarrow `pythonrun.c`. Its purpose is to turn a string into a Python module and then send it on to be executed. A Python module needs to have an entry-point, (`__main__`), to be executed as a standalone module. `PyRun_SimpleStringFlags()` creates the entry point implicitly.

Once `PyRun_SimpleStringFlags()` has created a module and a dictionary,

it calls `PyRun_StringFlags()`. `PyRun_SimpleStringFlags()` creates a fake filename and then calls the Python parser to create an AST from the string and return a module.

Note

Python modules are the data structure used to hand parsed code onto the compiler. The C structure for a Python module is `mod_ty` and is defined in `Include\Python-ast.h`.

Input with a Local Module

Another way to execute Python commands is by using the `-m` option with the name of a module. A typical example is `python -m unittest` to run the `unittest` module in the standard library.

Being able to execute modules as scripts was initially proposed in [PEP 338](#). The standard for explicit relative imports was defined in [PEP366](#).

The use of the “`-m`” flag implies that within the module package, you want to execute whatever is inside the entry point (`__main__`). It also implies that you want to search `sys.path` for the named module.

This search mechanism in the import library (`importlib`) is why you don’t need to remember where the `unittest` module is stored on your filesystem.

C^{Python} imports a standard library module, `runpy` and executes it using `PyObject_Call()`. The import is done using the C API function `PyImport_ImportModule()`, found within the `Python\import.c` file.

Note

In Python, if you had an object and wanted to get an attribute, you could call `getattr()`. In the C API, this call is `PyObject_GetAttrString()`, which is found in `Objects\object.c`. If you wanted to run a callable, you would give it parentheses, or you can run the `__call__()` property on any Python object. The `__call__()` method is implemented inside `Objects\object.c`:

```
>>> my_str = "hello world!"  
>>> my_str.upper()  
'HELLO WORLD!'  
>>> my_str.upper.__call__()  
'HELLO WORLD!'
```

The `runpy` module is written in pure Python and located in `Lib\runpy.py`.

Executing `python -m <module>` is equivalent to running `python -m runpy <module>`. The `runpy` module was created to abstract the process of locating and executing modules on an Operating System.

`runpy` does a few things to run the target module:

- Calls `__import__()` for the module name you provided
- Sets `__name__` (the module name) to a namespace called `__main__`
- Executes the module within the `__main__` namespace

The `runpy` module also supports executing directories and zip files.

Input From a Script File or Standard Input

If the first argument to `python` was a filename, such as `python test.py`, then CPython will open a filehandle, and pass the handle to `PyRun_SimpleFileExFlags()`, inside `Python\pythonrun.c`.

There are three paths this function can take:

1. If the file path is a `.pyc` file, it will call `run_pyc_file()`.
2. If the file path is a script file (`.py`) it will run `PyRun_FileExFlags()`.
3. If the file path is `stdin` because the user ran `command | python`, then treat `stdin` as a filehandle and run `PyRun_FileExFlags()`.

For `stdin` and basic script files, CPython will pass the filehandle to `PyRun_FileExFlags()` located in the `Python\pythonrun.c` file.

The purpose of `PyRun_FileExFlags()` is similar to `PyRun_SimpleStringFlags()`. CPython will load the filehandle into `PyParser_ASTFromFileObject()`.

Identical to `PyRun_SimpleStringFlags()`, once `PyRun_FileExFlags()` has created a Python module from the file, it sent it to `run_mod()` to be executed.

Input From Compiled Bytecode

If the user runs `python` with a path to a `.pyc` file, then instead of loading the file as a plain text file and parsing it, CPython will assume that the `.pyc` file contains a code object written to disk.

In `PyRun_SimpleFileExFlags()` there is a clause for the user providing a file path to a `.pyc` file.

The `run_pyc_file()` function inside `Python\pythonrun.c` marshals the code object from the `.pyc` file by using a filehandle. The code object data structure on the disk is the CPython compiler's way to cache compiled code so that it doesn't need to parse it every time the script is called.

Note

Marshaling is a term for copying the contents of a file into memory and converting them to a specific data structure.

Once the code object has been marshaled to memory, it is sent to `run_eval_code_obj()`, which calls `Python\ceval.c` to execute the code.

Conclusion

In this chapter, you've uncovered how Python's many configuration options are loaded and how code is input into the interpreter. Python's flexibility of input makes it a great tool for a range of applications, such as:

- Command-line utilities
- Long-running network applications, like web servers
- Short, composable scripts

Python's ability to set configuration properties in many ways causes complexity. For example, if you tested a Python application on Python 3.8, and it executed correctly. But in a different environment, it failed, you need to understand what settings were different in that environment. This means you'd need to inspect environment variables, runtime flags, and even the `sys config` properties. The compile-time properties found in `sys config` can be different amongst distributions of Python. For example, Python 3.8 downloaded from python.org for macOS has different default values than the Python 3.8 distribution found on Homebrew or the one found on the Anaconda distribution.

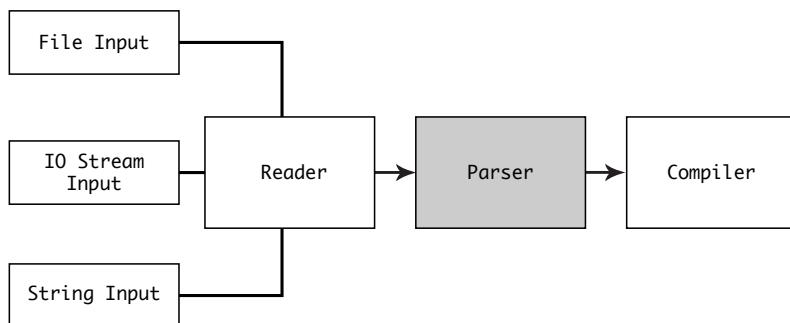
All of these input methods have an output of a Python module. In the next chapter, you will uncover how modules are created from the input.

[Leave feedback on this section »](#)

Lexing and Parsing with Syntax Trees

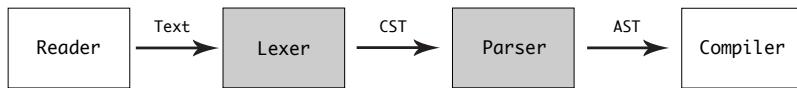
In the previous chapter, you explored how Python text is read from various sources. It needs to be converted into a structure that the compiler can use.

This stage is **parsing**:



In this chapter, you will explore how the text is parsed into logical structures that can be compiled.

There are two structures used to parse code in CPython, the **Concrete Syntax Tree** and the **Abstract Syntax Tree**.



The two parts of the parsing process are:

1. Creating a Concrete Syntax Tree using a **Parser-Tokenizer (Lexer)**
2. Creating an Abstract Syntax Tree from a Concrete Syntax Tree using a **Parser**

These two steps are common paradigms used in many programming languages.

Concrete Syntax Tree Generation

The Concrete Syntax Tree (CST) (or sometimes known as a **parse-tree**), is an ordered, rooted tree structure that represents code in a context-free grammar.

The CST is created from a **tokenizer** and **parser**. You explored the parser-generator in the chapter on **The Python Language and Grammar**. The output from the parser-generator is a Deterministic Finite Automaton (DFA) parsing table, describing the possible states of context-free grammar.

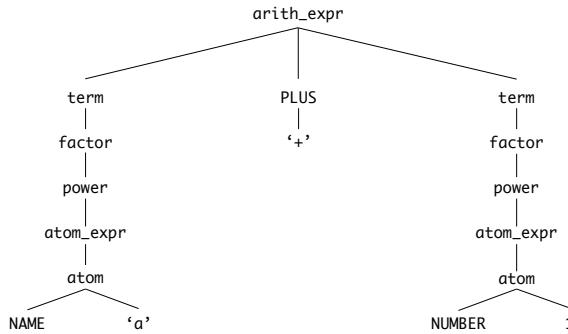
See Also

The original author of Python, Guido van Rossum is currently working on a contextual-grammar as an alternative to LL(1), the grammar used in CPython. The alternative is called Parser Expression Grammar, or PEG.

This will be available as an experimental feature in Python 3.9

In the grammar chapter, you explored some expression types, such as `if_stmt` and `with_stmt`. The Concrete Syntax Tree represents grammar symbols (like `if_stmt`) as branches, with tokens and terminals as leaf nodes.

For example, the arithmetic expression “`a + 1`” becomes the CST:



An arithmetic expression is represented here with three major branches, the left, operator, and right.

The parser iterates through tokens from an input stream and matches it against the possible states and tokens in the grammar to build a CST.

In `Grammar`, all of the symbols shown in the CST above are defined:

```

arith_expr: term (( '+' | '-' ) term)*
term: factor (( '*' | '@' | '/' | '%' | '//' ) factor)*
factor: ('+' | '-' | '~') factor | power
power: atom_expr [ '**' factor]
atom_expr: [AWAIT] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')') |
      '[' [testlist_comp] ']'
  
```

```
'{' [dictorsetmaker] '}' |  
NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
```

In [Grammar ▸ Tokens](#), the tokens are also defined:

```
ENDMARKER  
NAME  
NUMBER  
STRING  
NEWLINE  
INDENT  
DEDENT  
  
LPAR           '('  
RPAR           ')'  
LSQB           '['  
RSQB           ']'  
COLON          ':'  
COMMA          ','  
SEMI           ';'  
PLUS           '+'  
MINUS          '-'  
STAR           '*'  
...  
...
```

A `NAME` token represents the name of a variable, function, class, or module. Python's syntax doesn't allow a `NAME` to be:

- One of the reserved keywords, like `await` and `async`
- A numeric or other literal type

For example, if you tried to define a function named `1`, Python would raise a `SyntaxError`:

```
>>> def 1():  
  File "<stdin>", line 1  
    def 1():  
      ^  
SyntaxError: invalid syntax
```

A `NUMBER` is a particular token type to represent one of Python's many numeric values. Python has a special grammar for numbers, including:

- Octal values, e.g., `0o20`
- Hexadecimal values, e.g., `0x10`
- Binary values, e.g., `0b10000`
- Complex numbers, e.g., `10j`
- Floating-point numbers, e.g., `1.01`
- Underscores as commas, e.g., `1_000_000`

You can see compiled symbols and tokens using the `symbol` and `token` modules in Python:

```
$ ./python
>>> import symbol
>>> dir(symbol)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', '_main', '_name', '_value',
 'and_expr', 'and_test', 'annassign', 'arglist', 'argument',
 'arith_expr', 'assert_stmt', 'async_funcdef', 'async_stmt',
 'atom', 'atom_expr',
...
>>> import token
>>> dir(token)
['AMPERSHARP', 'AMPEREQUAL', 'AT', 'ATEQUAL', 'CIRCUMFLEX',
 'CIRCUMFLEXEQUAL', 'COLON', 'COMMA', 'COMMENT', 'DEDENT', 'DOT',
 'DOUBLESLASH', 'DOUBLESLASHEQUAL', 'DOUBLESTAR', 'DOUBLESTAREQUAL',
 ...]
```

The CPython Parser-Tokenizer

Programming languages have different implementations for the Lexer. Some use a Lexer-Generator, as a complement to the Parser-Generator.

CPython has a Parser/Tokenizer module, written in C.

Related Source Files

The source files relating to the parser-tokenizer are:

File	Purpose
Python \triangleright pythonrun.c	Executes the parser and the compiler from an input
Parser \triangleright parsetok.c	The Parser and Tokenizer implementation
Parser \triangleright tokenizer.c	Tokenizer implementation
Parser \triangleright tokenizer.h	Header file for the Tokenizer Implementation, describes data models like token state
Include \triangleright token.h	Declaration of token types, generated by <code>Tools</code> \triangleright <code>scripts</code> \triangleright <code>generate_token.py</code>
Include \triangleright node.h	Parse tree node interface and macros for the tokenizer

Inputting Data Into the Parser From a File

The entry point for the parser-tokenizer, `PyParser_ASTFromFileObject()`, takes a file handle, compiler flags and a `PyArena` instance and converts the file object into a module. There are two steps:

1. Convert to a CST using `PyParser_ParseFileObject()`
2. Convert into a AST/module, using the AST function `PyAST_FromNodeObject()`

The `PyParser_ParseFileObject()` function has two important tasks:

1. Instantiate a tokenizer state `tok_state` using `PyTokenizer_FromFile()`
2. Convert the tokens into a CST (a list of `node`) using `parsetok()`

Parser-Tokenizer Flow

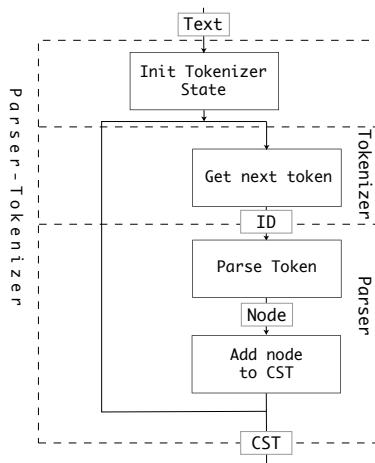
The parser-tokenizer takes text input and executes the tokenizer and parser in a loop until the cursor is at the end of the text (or a syntax error occurred).

Before execution, the parser-tokenizer establishes `tok_state`, a temporary data structure to store all state used by the tokenizer. The tokenizer state contains information such as the current cursor position and line.

The parser-tokenizer calls `tok_get()` to get the next token. The parser-tokenizer passes the resulting token ID to the parser, which uses the parser-generator DFA to create a node on the Concrete Syntax Tree.

`tok_get()` is one of the most complex functions in the whole CPython codebase. It has over 640 lines and includes decades of heritage with edge cases, new language features, and syntax.

The process of calling the tokenizer and parser in a loop can be shown as:



The CST root node returned by `PyParser_ParseFileObject()` is going to be essential for the next stage, converting a CST into an Abstract-Syntax-Tree (AST). The node type is defined in `Include\Node.h` as:

```

typedef struct _node {
    short          n_type;
    char          *n_str;
  
```

```
    int n_lineno;
    int n_col_offset;
    int n_nchildren;
    struct _node *n_child;
    int n_end_lineno;
    int n_end_col_offset;
}
```

Since the CST is a tree of syntax, token IDs, and symbols, it would be difficult for the compiler to make quick decisions based on the Python language.

Before you jump into the AST, there is a way to access the output from the parser stage. CPython has a standard library module `parser`, which exposes the C functions with a Python API.

The output will be in the numeric form, using the token and symbol numbers generated by the `make regen-grammar` stage, stored in `Include/token.h`:

```
[4, ''],
[0, '']]
```

To make it easier to understand, you can take all the numbers in the `symbol` and `token` modules, put them into a dictionary and recursively replace the values in the output of `parser.st2list()` with the names of the tokens:

```
cpython-book-samples▶21▶lex.py
```

```
import symbol
import token
import parser

def lex(expression):
    symbols = {v: k for k, v in symbol.__dict__.items()
               if isinstance(v, int)}
    tokens = {v: k for k, v in token.__dict__.items()
              if isinstance(v, int)}
    lexicon = {**symbols, **tokens}
    st = parser.expr(expression)
    st_list = parser.st2list(st)

    def replace(l: list):
        r = []
        for i in l:
            if isinstance(i, list):
                r.append(replace(i))
            else:
                if i in lexicon:
                    r.append(lexicon[i])
                else:
                    r.append(i)
        return r

    return replace(st_list)
```

You can run `lex()` with a simple expression, like `a + 1` to see how this is represented as a parser-tree:

In the output, you can see the symbols in lowercase, such as 'arith_expr' and the tokens in uppercase, such as 'NUMBER'.

Abstract Syntax Trees

The next stage in the CPython interpreter is to convert the CST generated by the parser into something more logical that can be executed.

Concrete Syntax Trees are a very literal representation of the text in the code file. At this stage, it could be a number of languages. Python's basic grammatical structure has been interpreted, but you could not use the CST to establish functions, scopes, loops or any of the core

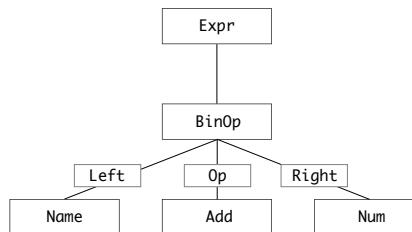
Python language features.

Before code is compiled, the CST needs to be converted into a higher-level structure that represents actual Python constructs. The structure is a representation of the CST, called an Abstract Syntax Tree (AST).

As an example, a binary operation in the AST is called a `BinOp` and is defined as a type of expression. It has three components:

- `left` - The left-hand part of the operation
- `op` - The operator, e.g., `+`, `-`, `*`
- `right` - The right-hand part of the expression

The AST for `a + 1` can be represented in an AST as:



ASTs are produced by the CPython parser process, but you can also generate them from Python code using the `ast` module in the Standard Library.

Before diving into the implementation of the AST, it would be useful to understand what an AST looks like for a simple piece of Python code.

Related Source Files

The source files relating to Abstract Syntax Trees are:

File	Purpose
Include ▶ Python-ast.h	Declaration of AST Node types, generated by Parser ▶ asdl_c.py
Parser ▶ Python.asdl	A list of AST Node Types and Properties in a domain-specific-language, ASDL 5
Python ▶ ast.c	The AST implementation

Using Instaviz to View Abstract Syntax Trees

Instaviz is a Python package written for this book. It displays ASTs and compiled code in a web interface.

To install `instaviz`, install the `instaviz` package from pip:

```
$ pip install instaviz
```

Then, open up a REPL by running `python` at the command line with no arguments. The function `instaviz.show()` takes a single argument of type `code object`.

You will cover code objects in the next chapter. For this example, define a function and use the name of the function as the argument value:

```
$ python
>>> import instaviz
>>> def example():
    a = 1
    b = a + 1
    return b

>>> instaviz.show(example)
```

You'll see a notification on the command-line that a web server has started on port 8080. If you were using that port for something else, you could change it by calling `instaviz.show(example, port=9090)` or another port number.

In the web browser, you can see the detailed breakdown of your function:

Code Object Properties

Field	Value
co_argcount	0
co_cellvars	0
co_code	64017d007c00640117007d017c015300
co_consts	(None, 1)
co_filename	test.py
co_firstlineno	4
co_freevars	0
co_kwonlyargcount	0
co_lnotab	b'\x00\x01\x04\x01\x08\x01'
co_name	foo
co_names	0
co_nlocals	2
co_stacksize	2
co_varnames	('a', 'b')

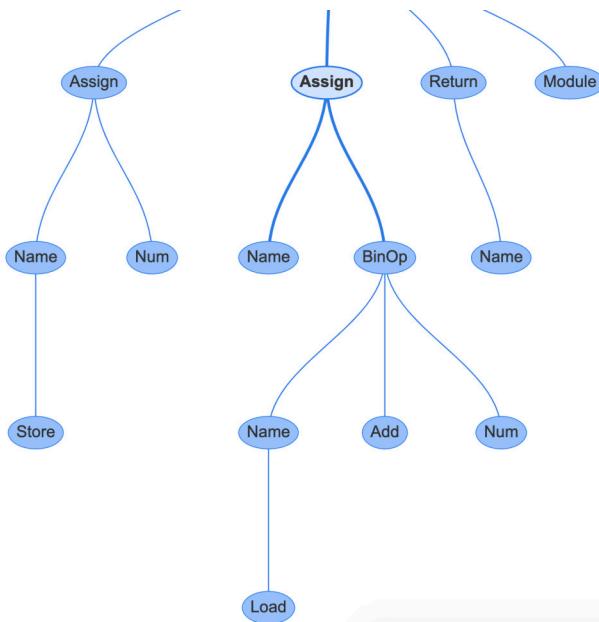
```
4 def foo():
5
6     b = a + 1
7     return b
```

Graph direction: [Up-Down](#) [Down-Up](#) [Left-Right](#) [Right-Left](#)

The bottom left graph is the function you declared in REPL, represented as an Abstract Syntax Tree. Each node in the tree is an AST type. They are found in the `ast` module, and all inherit from `_ast.AST`.

Some of the nodes have properties that link them to child nodes, unlike the CST, which has a generic child node property.

For example, if you click on the Assign node in the center, this links to the line `b = a + 1`:



The **assign** node has two properties:

1. **targets** is a list of names to assign. It is a list because you can assign to multiple variables with a single expression using unpacking
2. **value** is the value to assign, which in this case is a **BinOp** statement, $a + 1$.

If you click on the **BinOp** statement, it shows the properties of relevance:

- **left**: the node to the left of the operator
- **op**: the operator, in this case, an **Add** node (+) for addition
- **right**: the node to the right of the operator

Node Properties

Select a node on the AST graph to see properties.

json	object
targets	array
0	object
id : 'b'	string
ctx	object
value	object
left	object
id : 'a'	string
ctx	object
op	object
right	object
n : 1	number
lineno : 3	number

AST Compilation

Compiling an AST in C is not a straightforward task. The `Python>ast.c` module has over 5000 lines of code.

There are a few entry points, forming part of the AST's public API. The AST API takes a node tree (CST), a filename, the compiler flags, and a memory storage area. The result type is `mod_ty` representing a Python module, defined in `Include>Python-ast.h`.

`mod_ty` is a container structure for one of the five module types in Python:

1. Module
2. Interactive
3. Expression
4. FunctionType
5. Suite

The module types are all listed in `Parser>Python.asdl`. You will see the module types, statement types, expression types, operators, and comprehensions all defined in this file. The names of the types in `Parser>Python.asdl` relate to the classes generated by the AST and the same

classes named in the `ast` standard module library:

```
-- ASDL's 5 builtin types are:
-- identifier, int, string, object, constant

module Python
{
    mod = Module(stmt* body, type_ignore *type_ignores)
    | Interactive(stmt* body)
    | Expression(expr body)
    | FunctionType(expr* argtypes, expr returns)
```

The AST module imports `Include`▶`Python-ast.h`, a file created automatically from `Parser`▶`Python.asdl` when regenerating grammar. The parameters and names in `Include`▶`Python-ast.h` correlate directly to those specified in `Parser`▶`Python.asdl`.

The `mod_ty` type is generated into `Include`▶`Python-ast.h` from the `Module` definition in `Parser`▶`Python.asdl`:

```
enum _mod_kind {Module_kind=1, Interactive_kind=2, Expression_kind=3,
                FunctionType_kind=4, Suite_kind=5};

struct _mod {
    enum _mod_kind kind;
    union {
        struct {
            asdl_seq *body;
            asdl_seq *type_ignores;
        } Module;

        struct {
            asdl_seq *body;
        } Interactive;

        struct {
            expr_ty body;
        } Expression;
```

```

struct {
    asdl_seq *argtypes;
    expr_ty returns;
} FunctionType;

struct {
    asdl_seq *body;
} Suite;

} v;
};


```

The C header file and structures are there so that the `Python>ast.c` program can quickly generate the structures with pointers to the relevant data.

The AST entry-point, `PyAST_FromNodeObject()`, is essentially a `switch` statement around the result from `TYPE(n)`. `TYPE()` is a macro used by the AST to determine what type a node in the concrete syntax tree is. The result of `TYPE()` will be either a symbol or token type. By starting at the root node, it can only be one of the module types defined as `Module`, `Interactive`, `Expression`, `FunctionType`.

- For `file_input`, the type should be a `Module`
- For `eval_input`, such as from a REPL, the type should be an `Expression`

For each type of statement, there is a corresponding `ast_for_xxx` C function in `Python>ast.c`, which will look at the CST nodes to complete the properties for that statement.

One of the simpler examples is the power expression, i.e., `2**4` is 2 to the power of 4.

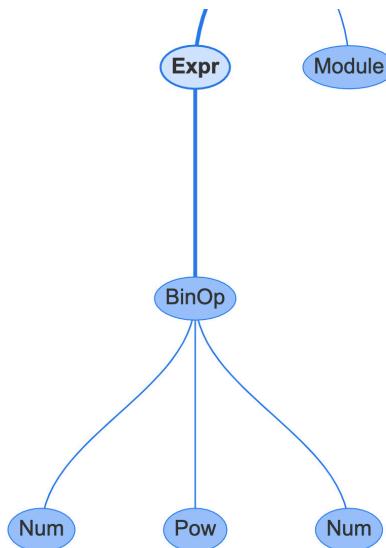
The `ast_for_power()` function will return a `BinOp` (binary operation) with the operator as `Pow` (power), the left hand of `e` (2), and the right hand of `f` (4):

Python ▶ `ast.c` line 2694

```
static expr_ty
ast_for_power(struct compiling *c, const node *n)
{
    /* power: atom trailer* ('**' factor)*
     */
    expr_ty e;
    REQ(n, power);
    e = ast_for_atom_expr(c, CHILD(n, 0));
    if (!e)
        return NULL;
    if (NCH(n) == 1)
        return e;
    if (TYPE(CHILD(n, NCH(n) - 1)) == factor) {
        expr_ty f = ast_for_expr(c, CHILD(n, NCH(n) - 1));
        if (!f)
            return NULL;
        e = BinOp(e, Pow, f, LINENO(n), n->n_col_offset,
                  n->n_end_lineno, n->n_end_col_offset, c->c_arena);
    }
    return e;
}
```

You can see the result of this if you send a short function to the `instaviz` module:

```
>>> def foo():
...     2**4
>>> import instaviz
>>> instaviz.show(foo)
```



In the UI, you can also see the corresponding properties:

Node Properties

Select a node on the AST graph to see properties.

json	object
value	object
left	object
n : 2	number
op	object
right	object
n : 4	number
lineno : 2	number

In summary, each statement type and expression has a corresponding `ast_for_*` function to create it. The arguments are defined in `Parser` ↪ `Python.asdl` and exposed via the `ast` module in the standard library. If an expression or statement has children, then it will call the corresponding `ast_for_*` child function in a depth-first traversal.

Important Terms to Remember

- **Concrete Syntax Tree CST** A non-contextual tree representation of tokens and symbols
- **Parse-Tree** Another term for Concrete Syntax Tree
- **Abstract Syntax Tree** A contextual tree representation of Python's grammar and statements
- **Token** A type of symbol, e.g., “+”
- **Tokenizer** The process of converting text into tokens
- **Parser** A generic term of the process in converting text into a CST or AST

Example: Adding an Almost Equal Comparison Operator

To bring all this together, you can add a new piece of syntax to the Python language and recompile CPython to understand it.

A **comparison expression** will compare the values of two or more values. For example,

```
>>> a = 1
>>> b = 2
>>> a == b
False
```

The operator used in the comparison expression is called the **comparison operator**. Here are some comparison operators you may recognize:

- < Less than
- > Greater than
- == Equal to
- != Not equal to

See Also

Rich comparisons in the data model were proposed for Python 2.1 in [PEP207](#). The PEP contains context, history, and justification for custom Python types to implement comparison methods.

We will now add another comparison operator, called **almost equal**, represented by `~=` with the following behaviors:

- If you compare a float and an integer, it will cast the float into an integer and compare the result.
- If you compare two integers, it will use the normal equality operators.

Once implemented, this new operator should return the following in a REPL:

```
>>> 1 ~= 1
True
>>> 1 ~= 1.0
True
>>> 1 ~= 1.01
True
>>> 1 ~= 1.9
False
```

To add the new operator, you first need to update the CPython grammar.

In [Grammar ▶ Grammar](#), the comparison operators are defined as a symbol, `comp_op`:

```
comp_op: '<' | '>' | '==' | '>=' | '<=' | '<>' | '!='
        | 'in' | 'not'
        | 'in' | 'is' | 'is' 'not'
```

Change this line to include the `~=` comparison operator in between `'!-'` and `'in'`:

Example: Adding an Almost Equal Comparison Operator

```
comp_op: '<' | '>' | '==' | '>=' | '<=' | '<>' | '!='
         | '~='
         | 'in' | 'is' | 'is' 'not'
```

To update the grammar and tokens in C, you now have to regenerate the headers:

On macOS/Linux:

```
$ make regen-all
```

On Windows, within the `PCBuild` directory:

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

The tokenizer will be automatically updated by these steps. For example, open the `Parser/token.c` source and see how a case in the `PyToken_TwoChars()` function has changed:

```
case '~=':
    switch (c2) {
        case '=': return ALMOSTEQUAL;
    }
    break;
}
```

If you recompile CPython at this stage and open a REPL, you can see that the tokenizer can successfully recognise the token, but the AST does not know how to handle it:

```
$ ./python
>>> 1 ~= 2
SystemError: invalid comp_op: ~=
```

This exception is raised by `ast_for_comp_op()` inside `Python ▶ ast.c` because it does not recognise `ALMOSTEQUAL` as a valid operator for a comparison statement.

`Compare` is an expression type defined in `Parser ▶ Python.asdl`, it has properties for the left expression, a list of operators, `ops`, and a list of expressions to compare to, `comparators`:

Example: Adding an Almost Equal Comparison Operator

```
| Compare(expr left, cmpop* ops, expr* comparators)
```

Inside the `Compare` definition is a reference to the `cmpop` enumeration:

```
cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn
```

This is a list of possible AST leaf nodes that can act as comparison operators. Ours is missing and needs to be added. Update the list of options to include a new type, `Ale` (**Almost Equal**):

```
cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn | Ale
```

Next, regenerate the AST again to update the AST C header files:

```
$ make regen-ast
```

This will have updated the comparison operator, `_cmpop`, enum inside `Include/Python-ast.h` to include the `Ale` option:

```
typedef enum _cmpop { Eq=1, NotEq=2, Lt=3, LtE=4, Gt=5, GtE=6, Is=7,
                      IsNot=8, In=9, NotIn=10, Ale=11 } cmpop_ty;
```

The AST has no knowledge that the `ALMOSTEQUAL` token is equivalent to the `Ale` comparison operator. So you need to update the C code for the AST.

Navigate to `ast_for_comp_op()` in `Python>ast.c`. Find the switch statement for the operator tokens. This returns one of the `_cmpop` enumeration values.

Add two lines, to catch the `ALMOSTEQUAL` token and return the `Ale` comparison operator:

Python>ast.c line 1199

```
static cmpop_ty
ast_for_comp_op(struct compiling *c, const node *n)
{
    /* comp_op: '<'/'>'/'=='/'>='/'<='/'!=/'in'/'not' 'in'/'is'
```

Example: Adding an Almost Equal Comparison Operator

```
    / 'is' 'not'
*/
REQ(n, comp_op);
if (NCH(n) == 1) {
    n = CHILD(n, 0);
    switch (TYPE(n)) {
        case LESS:
            return Lt;
        case GREATER:
            return Gt;
        case ALMOSTEQUAL: // Add this line to catch the token
            return A1E;    // And this one to return the AST node
    }
}
```

Now recompile CPython and open up a REPL to test the command:

```
>>> a = 1
>>> b = 1.0
>>> a ~= b
True
```

At this stage, the tokenizer and the AST can parse this code, but the compiler won't know how to handle the operator. To test the AST representation, use the `ast.parse()` function and explore the first operator in the expression:

```
>>> import ast
>>> m = ast.parse('1 ~= 2')
>>> m.body[0].value.ops[0]
<_ast.A1E object at 0x10a8d7ee0>
```

This is an instance of our `A1E` comparison operator type, so the AST has correctly parsed the code.

In the next chapter, you will learn about how the CPython compiler works, and revisit the almost-equal operator to build out its behavior.

Conclusion

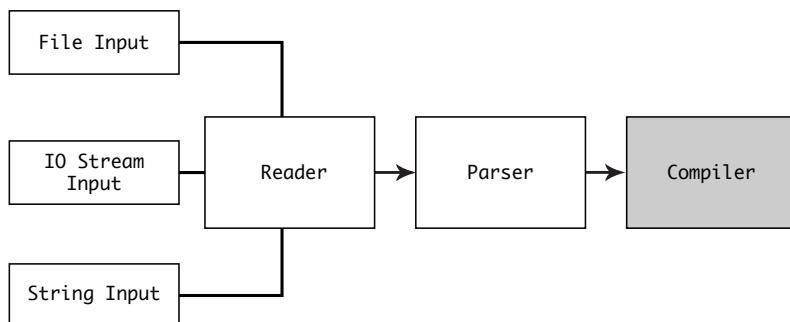
Cython's versatility and low-level execution API make it the ideal candidate for an embedded scripting engine. You will see Cython used in many UI applications, such as Game Design, 3D graphics, and system automation.

The interpreter process is flexible and efficient, and now you have an understanding of how it works you're ready to understand the compiler.

[Leave feedback on this section »](#)

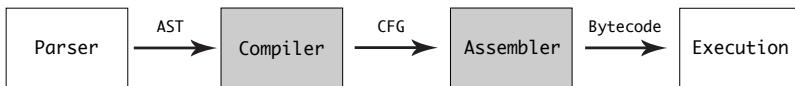
The Compiler

After completing the task of parsing, the interpreter has an AST with the operations, functions, classes, and namespaces of the Python code. The job of the **compiler** is to turn the AST into instructions the CPU could understand.



This compilation task is split into two components:

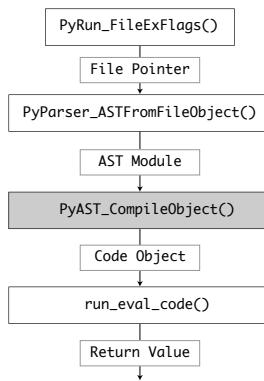
1. **Compiler** - Traverse the AST and create a **control-flow-graph** (CFG), which represents the logical sequence for execution
2. **Assembler** - Convert the nodes in the CFG to sequential, executable statements, known as **bytecode**



Important

Throughout this chapter, it is important to remember that the unit of compilation for CPython is a module. The compilation steps and process indicated in this chapter will happen once for each module in your project.

In this chapter, you will focus on the compilation of an AST module into a code object:



The `PyAST_CompileObject()` function is the main entry point to the CPython compiler. It takes a Python AST module as its primary argument, along with the name of the file, the globals, locals, and the PyArena all created earlier in the interpreter process.

Note

We're starting to get into the guts of the CPython compiler now, with decades of development and Computer Science theory behind it. Don't be put off by the size of it. Once you break down the compiler into logical steps, it is easier to understand.

Related Source Files

The source files relating to the compiler are:

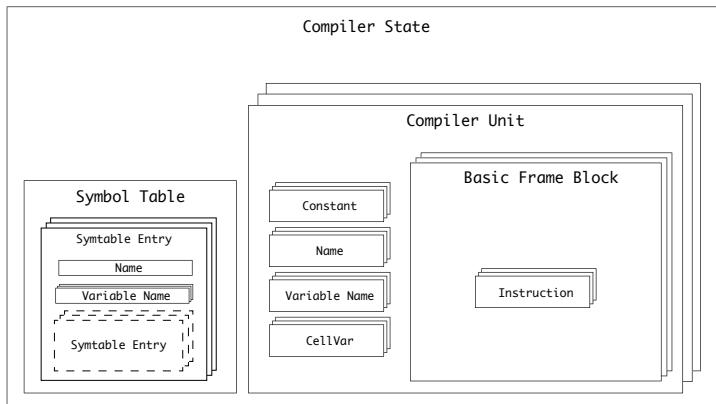
File	Purpose
Python\pythonrun.c	Executes the parser and the compiler from an input
Python\compile.c	The compiler implementation
Include\compile.h	The compiler API and type definitions

Important Terms

This chapter refers to many terms that may be new to you:

- The container type is the **compiler state**, which contains one **symbol table**
- The Symbol Table contains many **variable names** and can optionally contain child symbol tables
- The compiler type contains many **compiler units**
- Each compiler unit can contain many names, variable names, constants and cell variables
- A compiler unit contains many **basic frame blocks**
- Basic frame blocks many **bytecode instructions**

The compiler state container and its components can be shown as:



Instantiating a Compiler

Before the compiler starts, a global compiler state is created. The compiler state, (`compiler`), struct contains properties used by the compiler to such as compiler flags, the stack, and the `PyArena`. It also contains links to other data structures, like the symbol table.

Field	Type	Purpose
<code>c_filename</code>	<code>PyObject * (str)</code>	A string of the filename being compiled
<code>c_st</code>	<code>symtable *</code>	The compiler's symbol table
<code>c_future</code>	<code>PyFutureFeatures *</code>	A pointer to module's <code>__future__</code>
<code>c_flags</code>	<code>PyCompilerFlags *</code>	Inherited compiler flags (<i>See compiler flags</i>)
<code>c_optimize</code>	<code>int</code>	Optimization level
<code>c_interactive</code>	<code>int</code>	1 if in interactive mode
<code>c_nestlevel</code>	<code>int</code>	Current nesting level
<code>c_do_not_emit_bytecode</code>	<code>int</code>	The compiler won't emit any bytecode if this value is different from zero. This setting can be used to temporarily visit nodes without emitting bytecode to check only errors
<code>c_const_cache</code>	<code>PyObject * (dict)</code>	Python dict holding all constants, including names tuple

Field	Type	Purpose
u	compiler_unit*	Compiler state for current block
c_stack	PyObject * (list)	Python list holding compiler_unit ptrs
c_arena	PyArena *	A pointer to the memory allocation arena

Inside `PyAST_CompileObject()`, the compiler state is instantiated:

- If the module does not have a docstring (`__doc__`) property, an empty one is created here, as with the `__annotations__` property.
- `PyAST_CompileObject()` sets the filename in the compiler state to the value passed. This function is later used for stack traces and exception handling.
- The memory allocation arena for the compiler is set to the one used by the interpreter. See the Custom Memory Allocators section in the Memory Management chapter for more information on memory allocators.
- Any future flags are configured before the code is compiled.

Future Flags and Compiler Flags

Before the compiler runs, there are two types of flags to toggle the features inside the compiler. These come from two places:

1. The configuration state, which contains environment variables and command-line flags. *See the chapter on Configuration State.*
2. The use of `__future__` statements inside the source code of the module.

Future Flags

Future flags are required because of the syntax or features in that specific module. For example, Python 3.7 introduced delayed evaluation of type hints through the `annotations` future flag:

```
from __future__ import annotations
```

The code after this statement might use unresolved type hints, so the `__future__` statement is required. Otherwise, the module wouldn't import.

Reference of Future Flags in 3.9.0b1

As of 3.9.0b1, all but two of the future flags are mandatory and enabled automatically:

Import	Mandatory Since	Purpose
<code>nested_- scopes</code>	2.2	Statically nested scopes (PEP227)
<code>generators</code>	2.3	Simple generators (PEP255)
<code>division</code>	3.0	Use the “true” division operator (PEP238)
<code>absolute_- import</code>	3.0	Enable absolute imports (PEP328)
<code>with_- statement</code>	2.6	Enable the <code>with</code> statement (PEP343)
<code>print_- function</code>	3.0	Make <code>print</code> a function (PEP3105)
<code>unicode_- literals</code>	3.0	Make <code>str</code> literals Unicode instead of bytes (PEP3112)
<code>barry_as_- FLUFL</code>	N/A	Easter Egg (PEP401)
<code>generator_- stop</code>	3.7	Enable <code>StopIteration</code> inside generators (PEP479)
<code>annotations</code>	4.0	Postponed evaluation of type annotations (PEP563)

Note

Many of the `__future__` flags were used to aid portability between Python 2 and 3. As Python 4.0 approaches, you may see more future flags added.

Compiler Flags

The other compiler flags are specific to the environment, so they might change the way the code executes or the way the compiler runs, but they shouldn't link to the source in the same way that `__future__` statements do.

One example of a compiler flag would be the `-O` flag for optimizing the use of `assert` statements. This flag disables any `assert` statements, which may have been put in the code for debugging purposes. It can also be enabled with the `PYTHONOPTIMIZE=1` environment variable setting.

Symbol Tables

Before the code is compiled, a **symbol table** is created by the `PySymtable_BuildObject()` API.

The purpose of the symbol table is to provide a list of namespaces, globals, and locals for the compiler to use for referencing and resolving scopes.

Related Source Files

The source files relating to the symbol table are:

File	Purpose
Python \blacktriangleright <code>symtable.c</code>	The symbol table implementation
Include \blacktriangleright <code>symtable.h</code>	The symbol table API definition and type definitions
Lib \blacktriangleright <code>symtable.py</code>	The <code>symtable</code> standard library module

Symbol Table Data Structure

The `symtable` structure should be one `symtable` instance for the compiler, so namespacing becomes essential.

For example, if you create a method called `resolve_names()` in one class and declare another method with the same name in another class. Inside the module, you want to be sure which one is called.

The `symtable` serves this purpose, as well as ensuring that variables declared within a narrow scope don't automatically become globals.

The symbol table struct, (`symtable`), has the following fields:

Field	Type	Purpose
<code>st_filename</code>	<code>PyObject * (str)</code>	Name of file being compiled
<code>st_cur</code>	<code>_symtable_entry</code>	Current symbol table entry
*		
<code>st_top</code>	<code>_symtable_entry</code>	Symbol table entry for the module
*		
<code>st_blocks</code>	<code>PyObject * (dict)</code>	Map of AST node addresses to symbol table entries
<code>st_stack</code>	<code>PyObject * (list)</code>	Stack of namespace info
<code>st_global</code>	<code>PyObject * (dict)</code>	Reference to the symbols in <code>st_top</code> (<code>st_top->ste_symbols</code>)
<code>st_nblocks</code>	<code>int</code>	Number of blocks used
<code>st_private</code>	<code>PyObject * (str)</code>	Name of current class or NULL
<code>st_future</code>	<code>PyFutureFeatures</code>	Module's future features that affect the symbol table
*		
<code>recursion_depth</code>	<code>int</code>	Current recursion depth
<code>recursion_limit</code>	<code>int</code>	Recursion limit before <code>RecursionError</code> is raised. Set by <code>Py_SetRecursionLimit()</code>

Using the `symtable` Standard Library Module

Some of the symbol table C API is exposed in Python via [the `symtable` module](#) in the standard library.

Using another module called `tabulate` (available [on PyPi](#)), you can create a script to print a symbol table. Symbol tables can be nested, so if a module contains a function or class, that will have a symbol table.

Create a script called `symviz.py` with a recursive `show()` function:

cpython-book-samples▶30▶symviz.py

```
import tabulate
import symtable

code = """
def calc_pow(a, b):
    return a ** b
a = 1
b = 2
c = calc_pow(a,b)
"""

_st = symtable.symtable(code, "example.py", "exec")

def show(table):
    print("Symtable {0} ({1})".format(table.get_name(),
                                      table.get_type()))
    print(
        tabulate.tabulate(
            [
                (
                    symbol.get_name(),
                    symbol.is_global(),
                    symbol.is_local(),
                    symbol.get_namespaces(),
                )
                for symbol in table.get_symbols()
            ],
            headers=["name", "global", "local", "namespaces"],
            tablefmt="grid",
        )
    )
    if table.has_children():
        [show(child) for child in table.get_children()]

show(_st)
```

Run `symviz.py` at the command-line to see the symbol tables for the example code:

```
(venv) ➔ instaviz git:(master) ✘ python symviz.py
Symtable top (module)
+-----+-----+-----+-----+
| name | global | local  | namespaces |
+=====+=====+=====+=====+
| calc_pow | False  | True   | [<Function SymbolTable for calc_pow in example.py>] |
+-----+-----+-----+-----+
| a     | False  | True   | ()      |
+-----+-----+-----+-----+
| b     | False  | True   | ()      |
+-----+-----+-----+-----+
| c     | False  | True   | ()      |
+-----+-----+-----+-----+
Symtable calc_pow (function)
+-----+-----+-----+-----+
| name | global | local  | namespaces |
+=====+=====+=====+=====+
| a     | False  | True   | ()      |
+-----+-----+-----+-----+
| b     | False  | True   | ()      |
+-----+-----+-----+-----+
```

Symbol Table Implementation

The implementation of symbol tables is in `Python> symtable.c` and the primary interface is the `PySymtable_BuildObject()` function.

Similar to AST compilation covered in the last chapter, the `PySymtable_BuildObject()` function switches between the `mod_ty` possible types (Module, Expression, Interactive, Suite, FunctionType), and visits each of the statements inside them.

The Symbol Table will recursively explore the nodes and branches of the AST (of type `mod_ty`) and add entries to the symtable:

Python> `symtable.c` line 261

```
struct symtable *
PySymtable_BuildObject(mod_ty mod, PyObject *filename,
                      PyFutureFeatures *future)
{
    struct symtable *st = symtable_new();
```

```

asdl_seq *seq;
int i;
PyThreadState *tstate;
int recursion_limit = Py_GetRecursionLimit();
...

st->st_top = st->st_cur;
switch (mod->kind) {
case Module_kind:
    seq = mod->v.Module.body;
    for (i = 0; i < asdl_seq_LEN(seq); i++)
        if (!symtable_visit_stmt(st,
            (stmt_ty)asdl_seq_GET(seq, i)))
            goto error;
    break;
case Expression_kind:
...
case Interactive_kind:
...
case Suite_kind:
...
case FunctionType_kind:
...
}
...
}
}

```

For a module, `PySymtable_BuildObject()` will loop through each statement in the module and call `symtable_visit_stmt()`. The `symtable_visit_stmt()` is a huge `switch` statement with a case for each statement type (defined in `Parser`▶`Python.asdl`).

For each statement type, there is specific logic to that statement type. For example, a function definition (`FunctionDef_kind`) has particular logic for:

1. The current recursion depth against the recursion limit. If it has been exceeded a `RecursionError` is thrown.
2. The name of the function is added to the symbol table as a local

variable. In Python, functions are objects, so they can be passed as parameters or references.

3. Any non-literal default arguments to a function (non-keyword) are resolved from the symbol table.
4. Any non-literal default arguments to a function (keyword) are resolved from the symbol table.
5. Any type annotations for the arguments or the return type are resolved from the symbol table.
6. Any function decorators are resolved in sequence of definition.
7. The code block with the contents of the function is visited by `symtable_enter_block()`.
8. The arguments are visited and resolved.
9. The body of the function is visited and resolved.

Important

If you've ever wondered why Python's default arguments are mutable, the reason is in `symtable_visit_stmt()`. Argument defaults are a reference to the variable in the symtable. No extra work is done to copy any values to an immutable type.

As a preview, the C code for those steps in building a symtable for a function in `symtable_visit_stmt()`:

Python▶ `symtable.c` line 1171

```
static int
symtable_visit_stmt(struct symtable *st, stmt_ty s)
{
    if (++st->recursion_depth > st->recursion_limit) {
        PyErr_SetString(PyExc_RecursionError,
                       "maximum recursion depth exceeded during compilation");
        VISIT_QUIT(st, 0);
    }
}
```

```

switch (s->kind) {
  case FunctionDef_kind:
    if (!symtable_add_def(st, s->v.FunctionDef.name, DEF_LOCAL))
      VISIT_QUIT(st, 0);
    if (s->v.FunctionDef.args->defaults)
      VISIT_SEQ(st, expr, s->v.FunctionDef.args->defaults);
    if (s->v.FunctionDef.args->kw_defaults)
      VISIT_SEQ_WITH_NULL(st, expr,
                           s->v.FunctionDef.args->kw_defaults);
    if (!symtable_visit_annotations(st, s, s->v.FunctionDef.args,
                                    s->v.FunctionDef.returns))
      VISIT_QUIT(st, 0);
    if (s->v.FunctionDef.decorator_list)
      VISIT_SEQ(st, expr, s->v.FunctionDef.decorator_list);
    if (!symtable_enter_block(st, s->v.FunctionDef.name,
                             FunctionBlock, (void *)s, s->lineno,
                             s->col_offset))
      VISIT_QUIT(st, 0);
    VISIT(st, arguments, s->v.FunctionDef.args);
    VISIT_SEQ(st, stmt, s->v.FunctionDef.body);
    if (!symtable_exit_block(st, s))
      VISIT_QUIT(st, 0);
    break;
  case ClassDef_kind: {
    ...
  }
  case Return_kind:
    ...
  case Delete_kind:
    ...
  case Assign_kind:
    ...
  case AnnAssign_kind:
    ...
}

```

Once the resulting symbol table has been created, it is passed on to the compiler.

Core Compilation Process

Now that the `PyAST_CompileObject()` has a compiler state, a symtable, and a module in the form of the AST, the actual compilation can begin.

The purpose of the core compiler is to:

1. Convert the state, symtable, and AST into a [Control-Flow-Graph \(CFG\)](#)
2. Protect the execution stage from runtime exceptions by catching any logic and code errors

Accessing the Compiler From Python

You can call the compiler in Python by calling the built-in function `compile()`. It returns a `code` object:

```
>>> compile('b+1', 'test.py', mode='eval')
<code object <module> at 0x10f222780, file "test.py", line 1>
```

The same as with the `symtable()` API, a simple expression should have a mode of `'eval'`, and a module, function, or class should have a mode of `'exec'`.

The compiled code can be found in the `co_code` property of the code object:

```
>>> co.co_code
b'e\x00d\x00\x17\x00S\x00'
```

There is also a `dis` module in the standard library, which disassembles the bytecode instructions. You can print them on the screen, or get a list of `Instruction` instances.

Note

The `Instruction` type in the `dis` module is a reflection of the `instr` type in the C API.

If you import `dis` and give the `dis()` function the code object's `co_code` property it disassembles it and prints the instructions on the REPL:

```
>>> import dis
>>> dis.dis(co.co_code)
  0 LOAD_NAME           0 (0)
  2 LOAD_CONST          0 (0)
  4 BINARY_ADD
  6 RETURN_VALUE
```

`LOAD_NAME`, `LOAD_CONST`, `BINARY_ADD`, and `RETURN_VALUE` are all bytecode instructions. They're called bytecode because, in binary form, they were a byte long. However, since Python 3.6 the storage format was changed to a word, so now they're technically wordcode, not bytecode.

The [full list of bytecode instructions](#) is available for each version of Python, and it does change between versions. For example, in Python 3.7, some new bytecode instructions were introduced to speed up execution of specific method calls.

In earlier chapters, you explored the `instaviz` package. This included a visualization of the code object type by running the compiler. It also displays the bytecode operations inside the code objects.

Execute `instaviz` again to see the code object and bytecode for a function defined on the REPL:

```
>>> import instaviz
>>> def example():
    a = 1
    b = a + 1
    return b
>>> instaviz.show(example)
```

Compiler C API

The entry point for AST module compilation, `compiler_mod()`, switches to different compiler functions depending on the module type. If you assume that `mod` is a `Module`, the module is compiled into the

`c_stack` property as compiler units. Then `assemble()` is run to create a `PyCodeObject` from the compiler unit stack.

The new code object is returned back and sent on for execution by the interpreter, or cached and stored on disk as a `.pyc` file:

Python▶ `compile.c` line 1820

```
static PyCodeObject *
compiler_mod(struct compiler *c, mod_ty mod)
{
    PyCodeObject *co;
    int addNone = 1;
    static PyObject *module;
    ...
    switch (mod->kind) {
        case Module_kind:
            if (!compiler_body(c, mod->v.Module.body)) {
                compiler_exit_scope(c);
                return 0;
            }
            break;
        case Interactive_kind:
            ...
        case Expression_kind:
            ...
        case Suite_kind:
            ...
    ...
    co = assemble(c, addNone);
    compiler_exit_scope(c);
    return co;
}
```

The `compiler_body()` function loops over each statement in the module and visits it:

Python▶ `compile.c` line 1782

```

static int
compiler_body(struct compiler *c, asdl_seq *stmts)
{
    int i = 0;
    stmt_ty st;
    PyObject *docstring;
    ...
    for (; i < asdl_seq_LEN(stmts); i++)
        VISIT(c, stmt, (stmt_ty)asdl_seq_GET(stmts, i));
    return 1;
}

```

The statement type is determined through a call to the `asdl_seq_GET()` function, which looks at the AST node type.

Through a macro, `VISIT` calls a function in `Python>compile.c` for each statement type:

```

#define VISIT(C, TYPE, V) { \
    if (!compiler_visit_ ## TYPE((C), (V))) \
        return 0; \
}

```

For a `stmt` (the generic type for a statement) the compiler will then call `compiler_visit_stmt()` and switch through all of the potential statement types found in `Parser>Python.asdl`:

`Python>compile.c` line 3375

```

static int
compiler_visit_stmt(struct compiler *c, stmt_ty s)
{
    Py_ssize_t i, n;

    /* Always assign a lineno to the next instruction for a stmt. */
    c->u->u_lineno = s->lineno;
    c->u->u_col_offset = s->col_offset;
    c->u->u_lineno_set = 0;
}

```

```

switch (s->kind) {
    case FunctionDef_kind:
        return compiler_function(c, s, 0);
    case ClassDef_kind:
        return compiler_class(c, s);
    ...
    case For_kind:
        return compiler_for(c, s);
    ...
}

return 1;
}

```

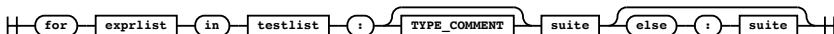
As an example, the For statement in Python is:

```

for i in iterable:
    # block
else: # optional if iterable is False
    # block

```

Or shown as a railroad diagram:



If the statement is a For type, `compiler_visit_stmt()` calls `compiler_for()`. There is an equivalent `compiler_*` function for all of the statement and expression types. The more straightforward types create the byte-code instructions inline, some of the more complex statement types call other functions.

Instructions

Many of the statements can have sub-statements. A for loop has a body, but you can also have complex expressions in the assignment and the iterator.

The compiler emits **blocks** to the compiler state. These blocks contain a sequence of instructions. The instruction data structure has an opcode, arguments, the target block (if this is a jump instruction), and the line number of the statement.

Instruction Type

The instruction type, `instr`, is defined as:

Field	Type	Purpose
<code>i_jabs</code>	<code>unsigned</code>	Flag to specify this is an absolute jump instruction
<code>i_jrel</code>	<code>unsigned</code>	Flag to specify this is a relative jump instruction
<code>i_opcode</code>	<code>unsigned char</code>	Opcode number this instruction represents (see <code>Include/Opcode.h</code>)
<code>i_oparg</code>	<code>int</code>	Opcode argument
<code>i_target</code>	<code>basicblock*</code>	Pointer to the <code>basicblock</code> target when <code>i_jrel</code> is true
<code>i_lineno</code>	<code>int</code>	Line number this instruction was created for

Jump Instructions

Jump instructions can either be absolute or relative. Jump instructions are used to “jump” from one instruction to another. Absolute jump instructions specify the exact instruction number in the compiled code object, whereas relative jump instructions specify the jump target relative to another instruction.

Basic Frame Blocks

A basic frame block (of type `basicblock`), contains the following fields:

Field	Type	Purpose
<code>b_list</code>	<code>basicblock *</code>	Each <code>basicblock</code> in a compilation unit is linked via <code>b_list</code> in the reverse order that the blocks are allocated
<code>b_iused</code>	<code>int</code>	Number of instructions used (<code>b_instr</code>)
<code>b_ialloc</code>	<code>int</code>	Length of instruction array (<code>b_instr</code>)
<code>b_instr</code>	<code>instr *</code>	Pointer to an array of instructions, initially <code>NULL</code>

Field	Type	Purpose
b_next	basicblock*	If b_next is non-NULL, it is a pointer to the next block reached by normal control flow
b_seen	unsigned	Used to perform a DFS of basicblocks. See assembly
b_return	unsigned	Is true if block returns a value (a RETURN_VALUE opcode is inserted)
b_-	int	Depth of stack upon entry of block, computed by stackdepth()
startdepth		
b_offset	int	Instruction offset for block, computed by assemble_jump_offsets()

Operations and Arguments

Depending on the type of operation, there are different arguments required. For example, ADDOP_JABS and ADDOP_JREL refer to “**ADD Operation with Jump to a RELative position**” and “**ADD Operation with Jump to an ABSolute position**”. The ADDOP_JREL and ADDOP_JABS macros which call `compiler_addop_j(struct compiler *c, int opcode, basicblock *b, int absolute)` and set the absolute argument to 0 and 1 respectively.

There are some other macros, like ADDOP_I calls `compiler_addop_i()` which add an operation with an integer argument, or ADDOP_O calls `compiler_addop_o()` which adds an operation with a `PyObject` argument.

Assembly

Once these compilation stages have completed, the compiler has a list of frame blocks, each containing a list of instructions and a pointer to the next block. The assembler performs a “depth-first-search” of the basic frame blocks and merges the instructions into a single bytecode sequence.

Assembler Data Structure

The assembler state struct, `assembler`, is declared in `Python>compile.c`.

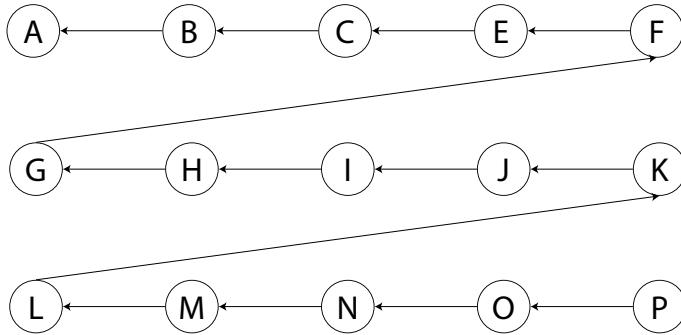
Field	Type	Purpose
a_bytecode	PyObject * (str)	String containing bytecode
a_offset	int	Offset into bytecode
a_nblocks	int	Number of reachable blocks
a_postorder	basicblock **	List of blocks in dfs postorder
a_lnotab	PyObject * (str)	String containing lnotab
a_lnotab_-	int	Offset into lnotab
off		
a_lineno	int	Last lineno of emitted instruction
a_lineno_-	int	Bytecode offset of last lineno
off		

Assembler Depth-First-Search Algorithm

The assembler uses a Depth-First-Search to traverse the nodes in the basic frame block graph. The DFS algorithm is not specific to CPython, but is a commonly used algorithm in graph traversal.

The CST and AST were both tree structures, whereas the compiler state is a graph structure, where the nodes are basic frame blocks containing instructions.

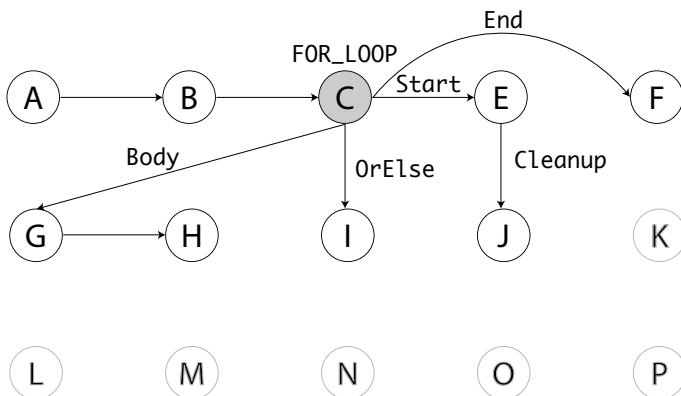
The basic frame blocks are linked by two graphs, one is in reverse order of creation (the `b_list` property of each block). A series of basic frame blocks arbitrarily named A-P would look like this:



The graph created from the `b_list` is used to sequentially visit every block in a compiler unit

The second graph uses the `b_next` property of each block. This list represents the control flow. Vertices in this graph are created by calls to `compiler_use_next_block(c, next)`, where `next` is the next block to draw a vertex to from the current block (`c->u->u_curblock`).

The `For` loop node graph might look something like this:



Both the sequential and control-flow graphs are used, but the control-flow graph is the one used by the DFS implementation.

Assembler C API

The assembler API has an entry point `assemble()`. The `assemble()` function has a few responsibilities:

- Calculate the number of blocks for memory allocation
- Ensure that every block that falls off the end returns `None` (This is why every function returns `None`, whether or not a `return` statement exists)
- Resolve any jump statements offsets that were marked as relative
- Call `dfs()` to perform a depth-first-search of the blocks
- Emit all the instructions to the compiler
- Call `makecode()` with the compiler state to generate the `PyCodeObject`

Python › `compile.c` line 6005

```
static PyCodeObject *
assemble(struct compiler *c, int addNone)
{
    ...
    if (!c->u->u_curblock->b_return) {
        NEXT_BLOCK(c);
        if (addNone)
            ADDOP_LOAD_CONST(c, Py_None);
        ADDOP(c, RETURN_VALUE);
    }
    ...
    dfs(c, entryblock, &a, nblocks);

    /* Can't modify the bytecode after computing jump offsets. */
    assemble_jump_offsets(&a, c);

    /* Emit code in reverse postorder from dfs. */
}
```

```

for (i = a.a_nbblocks - 1; i >= 0; i--) {
    b = a.a_postorder[i];
    for (j = 0; j < b->b_iused; j++)
        if (!assemble_emit(&a, &b->b_instr[j]))
            goto error;
}
...
co = makecode(c, &a);
error:
assemble_free(&a);
return co;
}

```

Depth-First-Search

The depth-first-search is performed by the `dfs()` function in `Python ▶ compile.c`, which follows the the `b_next` pointers in each of the blocks, marks them as seen by toggling `b_seen` and then adds them to the assemblers' `a_postorder` list in reverse order.

The function loops back over the assembler's post-order list and for each block, if it has a jump operation, recursively call `dfs()` for that jump:

Python ▶ `compile.c` line 5436

```

static void
dfs(struct compiler *c, basicblock *b, struct assembler *a, int end)
{
    int i, j;

    /* Get rid of recursion for normal control flow.
     * Since the number of blocks is limited, unused space in a_postorder
     * (from a_nbblocks to end) can be used as a stack for still not ordered
     * blocks. */
    for (j = end; b && !b->b_seen; b = b->b_next) {
        b->b_seen = 1;

```

```

        assert(a->a_nblocks < j);
        a->a_postorder[--j] = b;
    }

    while (j < end) {
        b = a->a_postorder[j++];
        for (i = 0; i < b->b_iused; i++) {
            struct instr *instr = &b->b_instr[i];
            if (instr->i_jrel || instr->i_jabs)
                dfs(c, instr->i_target, a, j);
        }
        assert(a->a_nblocks < j);
        a->a_postorder[a->a_nblocks++] = b;
    }
}

```

Once the assembler has assembled the graph into a CFG using DFS, the code object can be created.

Creating a Code Object

The task of `makecode()` is to go through the compiler state, some of the assembler's properties, and to put these into a `PyCodeObject` by calling `PyCode_New()`.

The variable names, constants are put as properties to the code object:

Python▶ `compile.c` line 5888

```

static PyCodeObject *
makecode(struct compiler *c, struct assembler *a)
{
    ...

    consts = consts_dict_keys_inorder(c->u->u_consts);
    names = dict_keys_inorder(c->u->u_names, 0);
    varnames = dict_keys_inorder(c->u->u_varnames, 0);
    ...

    cellvars = dict_keys_inorder(c->u->u_cellvars, 0);
}

```

```
...
    freevars = dict_keys_inorder(c->u->u_freevars,
                                PyTuple_GET_SIZE(cellvars));
...
    flags = compute_code_flags(c);
    if (flags < 0)
        goto error;

    bytecode = PyCode_Optimize(a->a_bytecode, consts,
                               names, a->a_lnotab);
...
    co = PyCode_NewWithPosOnlyArgs(
        posonlyargcount+posorkeywordargcount,
        posonlyargcount, kwonlyargcount, nlocals_int,
        maxdepth, flags, bytecode, consts, names,
        varnames, freevars, cellvars, c->c_filename,
        c->u->u_name, c->u->u_firstlineno, a->a_lnotab);
...
    return co;
}
```

You may also notice that the bytecode is sent to `PyCode_Optimize()` before it is sent to `PyCode_NewWithPosOnlyArgs()`. This function is part of the bytecode optimization process in `Python\peephole.c`.

The peephole optimizer goes through the bytecode instructions and in certain scenarios, replace them with other instructions. For example, there is an optimizer that removes any unreachable instructions that follow a `return` statement.

Using Instaviz to Show a Code Object

You can pull together all of the compiler stages with the `instaviz` module:

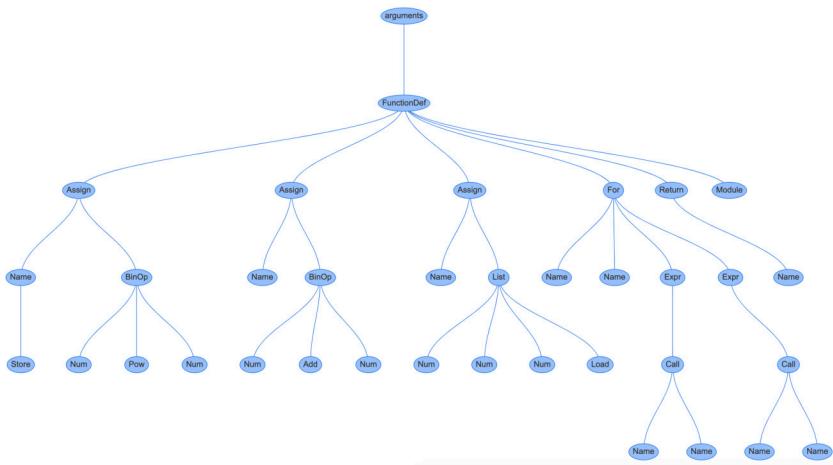
```
import instaviz

def foo():
```

```
a = 2**4
b = 1 + 5
c = [1, 4, 6]
for i in c:
    print(i)
else:
    print(a)
return c

instaviz.show(foo)
```

Will produce a large and complex AST graph:



You can see the bytecode instructions in sequence:

Using Instaviz to Show a Code Object

Disassembled Code

OpCode	Operation Name	Numeric Arg	Resolved Arg Value	Argument description	Index Offset	Starts Line	Is Jump Target?
100	LOAD_CONST	1	16	16	0	5	False
125	STORE_FAST	0	a	a	2	None	False
100	LOAD_CONST	2	6	6	4	6	False
125	STORE_FAST	1	b	b	6	None	False
100	LOAD_CONST	3	1	1	8	7	False
100	LOAD_CONST	4	4	4	10	None	False
100	LOAD_CONST	2	6	6	12	None	False
103	BUILD_LIST	3	3		14	None	False
125	STORE_FAST	2	c	c	16	None	False
120	SETUP_LOOP	28	48	to 48	18	8	False
124	LOAD_FAST	2	c	c	20	None	False
68	GET_ITER	None	None		22	None	False
93	FOR_ITER	12	38	to 38	24	None	True
125	STORE_FAST	3	i	i	26	None	False
116	LOAD_GLOBAL	0	print	print	28	9	False
124	LOAD_FAST	3	i	i	30	None	False
131	CALL_FUNCTION	1	1		32	None	False
1	POP_TOP	None	None		34	None	False
113	JUMP_ABSOLUTE	24	24		36	None	False
87	POP_BLOCK	None	None		38	None	True
116	LOAD_GLOBAL	0	print	print	40	11	False
124	LOAD_FAST	0	a	a	42	None	False
131	CALL_FUNCTION	1	1		44	None	False
1	POP_TOP	None	None		46	None	False
124	LOAD_FAST	2	c	c	48	12	True
83	RETURN_VALUE	None	None		50	None	False

The code object with the variable names, constants, and binary code:

Code Object Properties

Field	Value
co_argcount	0
co_cellvars	()
co_code	64017d0064027d0164036404640267037d02781c7c0244005d0c7d0374007c03830101007118570074007c00830101007c025300
co_consts	(None, 16, 6, 1, 4)
co_filename	test.py
co_firstlineno	4
co_freevars	()
co_kwonlyargcount	0
co_lnotab	b'\x00\x01\x04\x01\x04\x01\n\x01\x01\x01\x0c\x02\x08\x01'
co_name	foo
co_names	('print',)
co_nlocals	4
co_stacksize	3
co_varnames	('a', 'b', 'c', 'i')

Try it out with some other, more complex code that you have to learn more about CPython's compiler and code objects.

Example: Implementing the “Almost-Equal” Operator

After covering the compiler, bytecode instructions and the assembler, you can now modify CPython to support the “almost-equal” operator you compiled into the grammar in the last chapter.

First you have to add an internal `#define` for the `Py_ALE` operator, so it can be referenced inside the rich comparison functions for `PyObject`.

Open `Include/object.h`, and locate the following `#define` statements:

```
/* Rich comparison opcodes */
#define Py_LT 0
#define Py_LE 1
#define Py_EQ 2
#define Py_NE 3
#define Py_GT 4
#define Py_GE 5
```

Add an additional value, `PyALE` with a value of 6:

```
/* New Almost Equal comparator */
#define Py_ALE 6
```

Just underneath this expression is a macro `Py_RETURN_RICHCOMPARE`. Update this macro with a case statement for `Py_ALE`:

```
/*
 * Macro for implementing rich comparisons
 *
 * Needs to be a macro because any C-comparable type can be used.
 */
#define Py_RETURN_RICHCOMPARE(val1, val2, op)
    do {
        switch (op) {
            case Py_EQ: if ((val1) == (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE;
            case Py_NE: if ((val1) != (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE;
```

Example: Implementing the “Almost-Equal” Operator

```
    case Py_LT: if ((val1) < (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE;
    case Py_GT: if ((val1) > (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE;
    case Py_LE: if ((val1) <= (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE;
    case Py_GE: if ((val1) >= (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE;
/* + */ case Py_ALE: if ((val1) == (val2)) Py_RETURN_TRUE; Py_RETURN_FALSE;
    default:
        Py_UNREACHABLE();
    }
} while (0)
```

Inside `Objects/objects.c` there is a guard to check that the operator is within the range 0-5, because you added the value 6 you have to update that assertion:

`Objects/objects.c` line 709

```
PyObject *
PyObject_RichCompare(PyObject *v, PyObject *w, int op)
{
    PyThreadState *tstate = _PyThreadState_GET();

    assert(Py_LT <= op && op <= Py_GE);
```

Change that last line to:

```
assert(Py_LT <= op && op <= Py_ALE);
```

Next, you need to update the `COMPARE_OP` opcode to support `Py_ALE` as a value for the operator type.

First, edit `Objects/objects.c` and add `Py_ALE` into the `_Py_SwappedOp` list. This list is used for matching whether a custom class has one operator dunder method, but not the other.

For example, if you defined a class, `Coordinate`, you could define an equality operator by implementing the `__eq__` magic-method:

```
class Coordinate:
    def __init__(self, x, y):
```

Example: Implementing the “Almost-Equal” Operator

```
self.x = x
self.y = y

def __eq__(self, other):
    if isinstance(other, Coordinate):
        return (self.x == other.x and self.y == other.y)
    return super(self, other).__eq__(other)
```

Even though you haven’t implemented `__ne__` (not equal) for `Coordinate`, CPython assumes that the opposite of `__eq__` can be used.

```
>>> Coordinate(1, 100) != Coordinate(2, 400)
True
```

Inside `Objects/object.c`, locate the `_Py_SwappedOp` list and add `Py_EQ` to the end. Then add “`~`=” to the end of the `opstrings` list:

```
int _Py_SwappedOp[] = {Py_GT, Py_GE, Py_EQ, Py_NE, Py_LT, Py_LE, Py_EQ};

static const char * const opstrings[]
= {"<", "<=", "==" , "!=" , ">" , ">=" , "~="};
```

Open `Lib/opcode.py` and edit the list of rich comparison operators:

```
cmp_op = ('<', '<=', '==', '!=', '>', '>=', '~=')
```

Include the new operator at the end of the tuple:

```
cmp_op = ('<', '<=', '==', '!=', '>', '>=', '~=')
```

The `opstrings` list is used for error messages, if rich comparison operators are not implemented on a class.

Next, you can update the compiler to handle the case of a `PyCmp_EQ` property in a `BinOp` node.

Open `Python/compile.c` and find the `compiler_addcompare()` function:

Python/compile.c line 2479

Example: Implementing the “Almost-Equal” Operator

```
static int compiler_addcompare(struct compiler *c, cmpop_ty op)
{
    int cmp;
    switch (op) {
    case Eq:
        cmp = Py_EQ;
        break;
    case NotEq:
        cmp = Py_NE;
        break;
    case Lt:
        cmp = Py_LT;
        break;
    case LtE:
        cmp = Py_LE;
        break;
    case Gt:
        cmp = Py_GT;
        break;
    case GtE:
        cmp = Py_GE;
        break;
    }
```

Next, add another `case` to this switch statement to pair the `A1E` AST `comp_op` enumeration with the `PyCmp_A1E` opcode comparison enumeration:

Next, recompile CPython and open up a REPL. You should see the almost-equal operator behave in the same way as the `==` operator:

```
$ ./python
>>> 1 ~= 2
False
>>> 1 ~= 1
True
>>> 1 ~= 1.01
False
```

You can now program the behaviour of almost-equal to match the following scenario:

- `1 == 2` is `False`
- `1 == 1.01` is `True`, using floor-rounding

We can achieve this with some additional code. For now, you will cast both floats into integers and compare them.

CPython’s API has many functions for dealing with PyLong (`int`) and PyFloat (`float`) types. This will be covered in the chapter on Objects and Types.

Locate the `float_richcompare()` in `Objects/floatobject.c`, and under the `Compare:` goto definition add the following case:

`Objects/floatobject.c` line 358

```
static PyObject*
float_richcompare(PyObject *v, PyObject *w, int op)
{
    ...
    case Py_GT:
        r = i > j;
        break;
    /* New Code START */
    case Py_Ale: {
        double diff = fabs(i - j);
        double rel_tol = 1e-9; // relative tolerance
        double abs_tol = 0.1; // absolute tolerance
        r = (((diff <= fabs(rel_tol * j)) ||
              (diff <= fabs(rel_tol * i))) ||
              (diff <= abs_tol));
    }
    break;
}
/* New Code END */
return PyBool_FromLong(r);
```

This code will handle comparison of floating point numbers where the almost-equal operator has been used. It uses similar logic to `math.isclose()`, defined in [PEP485](#), but with a hardcoded absolute tolerance of 0.1.

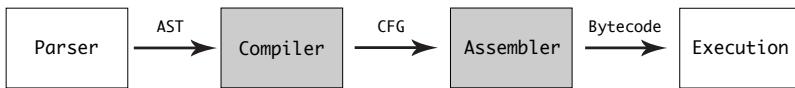
After recompiling CPython again, open a REPL and test it out:

```
$ ./python
>>> 1.0 ~= 1.01
True
>>> 1.02 ~= 1.01
True
>>> 1.02 ~= 2.01
False
>>> 1 ~= 1.01
True
>>> 1 ~= 1
True
>>> 1 ~= 2
False
>>> 1 ~= 1.9
False
>>> 1 ~= 2.0
False
>>> 1.1 ~= 1.101
True
```

In later chapters you will extend this implementation across other types.

Conclusion

In this chapter, you've explored how a parsed Python module is converted into a symbol table, a compilation state, and then a series of bytecode operations.



It is now the job of the CPython interpreter's core evaluation loop to execute those modules.

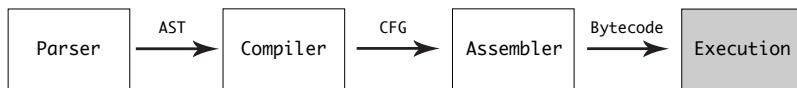
In the next chapter, you will explore how code objects are executed.

[Leave feedback on this section »](#)

The Evaluation Loop

So far, you have seen how Python code is parsed into an Abstract Syntax Tree and compiled into code objects. These code objects contain lists of discreet operations in the form of bytecode. There is one major thing missing for these code objects to be executed and come to life. They need input. In the case of Python, these inputs take the form of local and global variables. In this chapter, you will be introduced to a concept called a Value Stack, where variables are created, modified, and used by the bytecode operations in your compiled code objects.

Execution of code in CPython happens within a central loop, called the “evaluation loop.” The CPython interpreter will evaluate and execute a code object, either fetched from the marshaled .pyc file, or the compiler:



In the evaluation loop, each of the bytecode instructions is taken and executed using a [“Stack Frame” based system](#).

Note

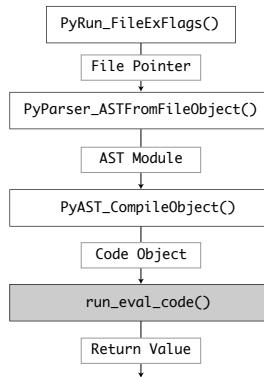
Stack Frames

Stack Frames are a data type used by many runtimes, not just Python. Stack Frames allow functions to be called and variables to be returned between functions. Stack Frames also contain arguments, local variables, and other stateful information.

A Stack Frame exists for every function call, and they are stacked in sequence. You can see CPython's frame stack anytime an exception is unhandled:

```
Traceback (most recent call last):
  File "example_stack.py", line 8, in <module>  <--- Frame
    function1()
  File "example_stack.py", line 5, in function1 <--- Frame
    function2()
  File "example_stack.py", line 2, in function2 <--- Frame
    raise RuntimeError
RuntimeError
```

When exploring the CPython compiler, you broke out just before the call to `run_eval_code_obj()`. In this next chapter, you will explore the interpreter API:



Related Source Files

The source files relating to the evaluation loop are:

File	Purpose
<code>Python/ceval.c</code>	The core evaluation loop implementation
<code>Python/ceval-gil.h</code>	The GIL definition and control algorithm

Important Terms

- The evaluation loop will take a **code object** and convert it into a series of **frame objects**
- The interpreter has at least one **thread**
- Each thread has a **thread state**
- Frame Objects are executed in a stack, called the **frame stack**
- Variables are referenced in a **value stack**

Constructing Thread State

Before a frame can be executed, it needs to be linked to a thread. CPython can have many threads running at any one time within a

single interpreter. The **interpreter state** includes a linked-list of those threads.

CPython always has at least one thread and each thread has it's own state.

See Also

Threading is covered in more detail within the “Parallelism and Concurrency” chapter.

Thread State Type

The thread state type, `PyThreadState` has over 30 properties, including:

- A unique identifier
- A linked-list to the other thread states
- The interpreter state it was spawned by
- The currently executing frame
- The current recursion depth
- Optional tracing functions
- The exception currently being handled
- Any async exception currently being handled
- A stack of exceptions raised, when multiple exceptions have been raised (e.g. `raise` within an `except` block)
- A GIL counter
- Async generator counters

Related Source Files

The source files relating to the thread state are spread across many files:

File	Purpose
Python\thread.c	The thread API implementation
Include\threadstate.h	Some of the thread state API and types definition
Include\pystate.h	The interpreter state API and types definition
Include\pythread.h	The threading API
Include\cpython\pystate.h	Some of the thread and interpreter state API

Constructing Frame Objects

Compiled code objects are inserted into frame objects. Frame objects are a Python type, so they can be referenced from C, and from Python. Frame objects also contain other runtime data that is required for executing the instructions in the code objects. This data includes the local variables, global variables and builtin modules.

Frame Object Type

The frame object type is a `PyObject` with the following additional properties:

Field	Type	Purpose
<code>f_back</code>	<code>PyFrameObject *</code>	Pointer to the previous in the stack, or <code>NULL</code> if first frame
<code>f_code</code>	<code>PyCodeObject *</code>	Code Object to be executed
<code>f_builtins</code>	<code>PyObject * (dict)</code>	Symbol table for the <code>builtin</code> module
<code>f_globals</code>	<code>PyObject * (dict)</code>	global symbol table (<code>PyDictObject</code>)
<code>f_locals</code>	<code>PyObject *</code>	Local symbol table (any mapping)
<code>f_valuestack</code>	<code>PyObject **</code>	Pointer to the last local
<code>f_stacktop</code>	<code>PyObject **</code>	Next free slot in <code>f_valuestack</code>
<code>f_trace</code>	<code>PyObject *</code>	Pointer to a custom tracing function. See section on frame tracing
<code>f_trace_lines</code>	<code>char</code>	Toggle for the custom tracing function to trace at line-level
<code>f_trace_-</code>	<code>char</code>	Toggle for the custom tracing function to trace at an opcode level
<code>opcodes</code>	<code>PyObject *</code>	Borrowed reference to a generator, or <code>NULL</code>
<code>f_gen</code>	<code>PyObject *</code>	Borrowed reference to a generator, or <code>NULL</code>

Field	Type	Purpose
<code>f_lasti</code>	int	Last instruction, if called
<code>f_lineno</code>	int	Current line number
<code>f_iblock</code>	int	Index of this frame in <code>f_blockstack</code>
<code>f_executing</code>	char	Flag whether the frame is still executing
<code>f_blockstack</code>	<code>PyTryBlock[]</code>	Sequence of <code>for</code> , <code>try</code> , and <code>loop</code> blocks
<code>f_localsplus</code>	<code>PyObject *[]</code>	Union of <code>locals</code> + <code>stack</code>

Related Source Files

The source files relating to frame objects are:

File	Purpose
<code>Objects\frameobject.c</code>	The frame object implementation and Python API
<code>Include\frameobject.h</code>	The frame object API and type definition

Frame Object Initialization API

The API for Frame Object Initialization, `PyEval_EvalCode()` is the entry point for evaluating a code object

`PyEval_EvalCode()` is a wrapper around the internal function `_PyEval_EvalCode()`.

Note

`_PyEval_EvalCode()` is a complex function that defines many behaviours of both frame objects and the interpreter loop. It is an important function to understand as it can also teach you some principles of the CPython interpreter design.

In this section you will step through the logic in `_PyEval_EvalCode()`.

The `_PyEval_EvalCode()` function specifies many arguments:

- **tstate:** a `PyThreadState *` pointing to the thread state of the thread this code will be evaluated on

- **_co:** a `PyCodeObject*` containing the code to be put into the frame object
- **globals:** a `PyObject*` (`dict`) with variable names as keys and their values
- **locals:** a `PyObject*` (`dict`) with variable names as keys and their values

Note

In Python, local and global variables stored as a dictionary. You can access this dictionary with the builtin functions `locals()` and `globals()`:

```
>>> a = 1
>>> print(locals()['a'])
1
```

The other arguments are optional, and not used for the basic API:

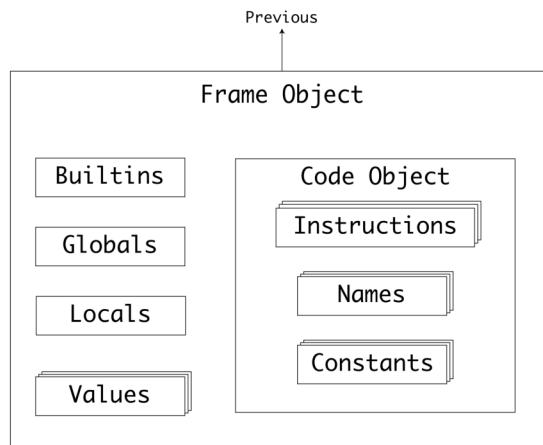
- **args:** a `PyObject*` (`tuple`) with positional argument values in order, and `argcount` for the number of values
- **kwnames:** a list of keyword argument names
- **kwargs:** a list of keyword argument values, and `kwcount` for the number of them
- **defs:** a list of default values for positional arguments, and `defcount` for the length
- **kwdefs:** a dictionary with the default values for keyword arguments
- **closure:** a tuple with strings to merge into the code objects `co_freevars` field
- **name:** the name for this evaluation statement as a string
- **qualname:** the qualified name for this evaluation statement as a string

The call to `_PyFrame_New_NoTrack()` creates a new frame. This API is

also available from the C API using `PyFrame_New()`. The `_PyFrame_New_NoTrack()` function will create a new `PyFrameObject` by following these steps:

1. Set the frame `f_back` property to the thread state's last frame
2. Load the current builtin functions by setting the `f_builtins` property and loading the `builtins` module using `PyModule_GetDict()`
3. Set the `f_code` property to the code object being evaluated
4. Set the `f_valuestack` property to the
5. Set the `f_stacktop` pointer to `f_valuestack`
6. Set the `global` property, `f_globals`, to the `globals` argument
7. Set the `locals` property, `f_locals`, to a new dictionary
8. Set the `f_lineno` to the code object's `co_firstlineno` property so that tracebacks contain line numbers
9. Set all the remaining properties to their default values

With the new `PyFrameObject` instance, the arguments to the frame object can be constructed:



Converting Keyword Parameters to a Dictionary

Function definitions can contain a `**kwargs` catch-all for keyword-arguments, for example:

```
def example(arg, arg2=None, **kwargs):
    print(kwargs['x'], kwargs['y'])  # this would resolve to a dictionary key
example(1, x=2, y=3)  # 2 3
```

In this scenario, a new dictionary is created, and the unresolved arguments are copied across. The `kwargs` name is then set as a variable in the local scope of the frame.

Converting Positional Arguments Into Variables

Each of the positional arguments (if provided) are set as local variables: In Python, function arguments are already local variables within the function body. When a positional argument is defined with a value, it is available within the function scope:

```
def example(arg1, arg2):
    print(arg1, arg2)
example(1, 2)  # 1 2
```

The reference counter for those variables is incremented, so the garbage collector won't remove them until the frame has evaluated (e.g. the function has finished and returned).

Packing Positional Arguments Into `*args`

Similar to `**kwargs`, a function argument prepended with a `*` can be set to catch all remaining positional arguments. This argument is a tuple and the `*args` name is set as a local variable:

```
def example(arg, *args):
    print(arg, args[0], args[1])
example(1, 2, 3)  # 1 2 3
```

Loading Keyword Arguments

If the function was called with keyword arguments and values, a dictionary is filled with any remaining keyword arguments passed by the caller that doesn't resolve to named arguments or positional arguments.

For example, the `e` argument was neither positional or named, so it is added to `**remaining`:

```
>>> def my_function(a, b, c=None, d=None, **remaining):
    print(a, b, c, d, remaining)

>>> my_function(a=1, b=2, c=3, d=4, e=5)
(1, 2, 3, 4, {'e': 5})
```

Note

Positional-only arguments are a new feature in Python 3.8.

Introduced in [PEP570](#), positional-only arguments are a way of stopping users of your API from using positional arguments with a keyword syntax.

For example, this simple function converts Fahrenheit to Celsius. Note, the use of `/` as a special argument separates positional-only arguments from the other arguments.

```
def to_celsius(fahrenheit, /, options=None):  
    return (fahrenheit-32)*5/9
```

All arguments to the left of `/` must be called only as a positional argument, and arguments to the right can be called as either positional or keyword arguments:

```
>>> to_celsius(110)
```

Calling the function using a keyword argument to a positional-only argument will raise a `TypeError`:

```
>>> to_celsius(fahrenheit=110)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: to_celsius() got some positional-only arguments  
passed as keyword arguments: 'fahrenheit'
```

The resolution of the keyword argument dictionary values comes after the unpacking of all other arguments. The PEP570 positional-only arguments are shown by starting the keyword-argument loop at `co_posonlyargcount`. If the `/` symbol was used on the 3rd argument, the value of `co_posonlyargcount` would be 2. `PyDict_SetItem()` is called for each remaining argument for adding it to the `locals` dictionary. When executing, each of the keyword arguments become scoped local variables.

If a keyword argument is defined with a value, that is available within this scope:

```
def example(arg1, arg2, example_kwarg=None):  
    print(example_kwarg)  # example_kwarg is already a local variable.
```

Adding Missing Positional Arguments

Any positional arguments provided to a function call that are not in the list of positional arguments are added to a `*args` tuple if this tuple does not exist, an exception is raised.

Adding Missing Keyword Arguments

Any keyword arguments provided to a function call that are not in the list of named keyword arguments are added to a `**kwargs` dictionary if this dictionary does not exist, an exception is raised.

Collapsing Closures

Any closure names are added to the code object's list of free variable names.

Creating Generators, Coroutines, and Asynchronous Generators

If the evaluated code object has a flag that it is a generator, coroutine, or async generator, then a new frame is created using one of the unique methods in the Generator, Coroutine, or Async libraries and the current frame is added as a property.

See Also

The APIs and implementation of generators, coroutines, and async frames are covered in the chapter on parallelism and concurrency

The new frame is then returned, and the original frame is not evaluated. The frame is only evaluated when the generator/coroutine/async method is called on to execute its target.

Lastly, `_PyEval_EvalFrame()` is called with the new frame.

Frame Execution

As covered earlier in the compiler and AST chapters, the code object contains a binary encoding of the bytecode to be executed. It also contains a list of variables and a symbol table.

The local and global variables are determined at runtime based on how that function, module, or block was called. This information is added to the frame by the `_PyEval_EvalCode()` function. There are other uses of frames, like the coroutine decorator, which dynamically generates a frame with the target as a variable.

The public API, `PyEval_EvalFrameEx()` calls the interpreter's configured frame evaluation function in the `eval_frame` property. Frame evaluation was [made pluggable in Python 3.7 with PEP 523](#).

`_PyEval_EvalFrameDefault()` is the default function and the only option bundled with CPython.

Frames are executed in the main execution loop inside `_PyEval_EvalFrameDefault()`. This central function brings everything together and brings your code to life. It contains decades of optimization since even a single line of code can have a significant impact on performance for the whole of CPython.

Everything that gets executed in CPython goes through this function.

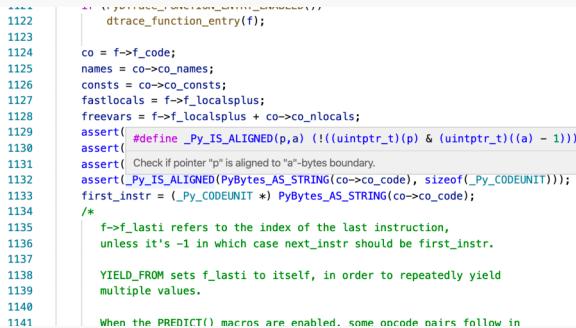
Note

Something you might notice when reading `Python.h` `ceval.c`, is how many times C macros have been used. C Macros are a way of having reusable code without the overhead of making function calls. The compiler converts the macros into C code and then compile the generated code.

If you want to see the expanded code, you can run `gcc -E` on Linux and macOS:

```
$ gcc -E Python/ceval.c
```

In [Visual Studio Code](#), inline macro expansion shows once you have installed the official C/C++ extension:



```

1122     dtrace_function_entry(f);
1123
1124     co = f->f_code;
1125     names = co->co_names;
1126     consts = co->co_consts;
1127     fastlocals = f->f_localsplus;
1128     freevars = f->f_localsplus + co->co_nlocals;
1129     assert( #define _Py_IS_ALIGNED(p,a) (!((uintptr_t)(p) & (uintptr_t)((a) - 1)))
1130             assert( Check if pointer "p" is aligned to "a"-bytes boundary.
1131             assert(_Py_IS_ALIGNED(PyBytes_AS_STRING(co->co_code), sizeof(_Py_CODEUNIT)));
1132             first_instr = (_Py_CODEUNIT *) PyBytes_AS_STRING(co->co_code);
1133             /*
1134             f->f_lasti refers to the index of the last instruction,
1135             unless it's -1 in which case next_instr should be first_instr.
1136
1137             YIELD_FROM sets f_lasti to itself, in order to repeatedly yield
1138             multiple values.
1139
1140             When the PREDICT() macros are enabled, some oncode pairs follow in

```

In CLion, select a macro and press `Alt + Space` to peek into its definition.

Frame Execution Tracing

You can step through frame execution in Python 3.7 and beyond by enabling the tracing attribute on the current thread. In the `PyFrameObject` type, there is a `f_trace` property, of type `PyObject *`. The value is expected to point to a Python function.

This code example sets the global tracing function to a function called

`my_trace()` that gets the stack from the current frame, prints the disassembled opcodes to the screen, and some extra information for debugging:

```
cpython-book-samples▶31▶my_trace.py
```

```
import sys
import dis
import traceback
import io

def my_trace(frame, event, args):
    frame.f_trace_opcodes = True
    stack = traceback.extract_stack(frame)
    pad = "    "*len(stack) + "|"
    if event == 'opcode':
        with io.StringIO() as out:
            dis.disco(frame.f_code, frame.f_lasti, file=out)
            lines = out.getvalue().split('\n')
            [print(f"{pad}{l}") for l in lines]
    elif event == 'call':
        print(f"{pad}Calling {frame.f_code}")
    elif event == 'return':
        print(f"{pad}Returning {args}")
    elif event == 'line':
        print(f"{pad}Changing line to {frame.f_lineno}")
    else:
        print(f"{pad}{frame} ({event} - {args})")
    print(f"{pad}-----")
    return my_trace

sys.settrace(my_trace)

# Run some code for a demo
eval('"-'.join([letter for letter in "hello"]))
```

The `sys.settrace()` function will set the current thread state default tracing function to the one provided. Any new frames created after this call will have `f_trace` set to this function.

This code snippet prints the code within each stack and points to the next operation before it is executed. When a frame returns a value, the return statement is printed:

```
→ cpython git:(master) ✘ ./python.exe my_trace.py
| Calling <code object <module> at 0x104cd110, file "<string>", line 1>
| Changing line to 1
| -----
| 1 → 0 LOAD_CONST          0 ('-')
| 2 LOAD_METHOD            0 (join)
| 4 LOAD_CONST            1 (<code object <listcomp> at 0x104cdce0, file "<string>", line 1>
| 6 LOAD_CONST            2 ('<listcomp>')
| 8 MAKE_FUNCTION          0
| 10 LOAD_CONST           3 ('hello')
| 12 GET_ITER
| 14 CALL_FUNCTION          1
| 16 CALL_METHOD            1
| 18 RETURN_VALUE

→ 1 0 LOAD_CONST          0 ('-')
  2 LOAD_METHOD            0 (join)
  4 LOAD_CONST            1 (<code object <listcomp> at 0x104cdce0, file "<string>", line 1>
  6 LOAD_CONST            2 ('<listcomp>')
  8 MAKE_FUNCTION          0
  10 LOAD_CONST           3 ('hello')
  12 GET_ITER
  14 CALL_FUNCTION          1
  16 CALL_METHOD            1
  18 RETURN_VALUE
```

The full list of possible bytecode instructions is available on the [dis module](#) documentation.

The Value Stack

Inside the core evaluation loop, a value stack is created. This stack is a list of pointers to `PyObject` instances. These could be values like variables, references to functions (which are objects in Python), or any other Python object.

Bytecode instructions in the evaluation loop will take input from the value stack.

Example Bytecode Operation, `BINARY_OR`

The binary operations that you have been exploring in previous chapters compile into a single instruction.

If you inserted an `or` statement in Python:

```
if left or right:
    pass
```

The `or` operation would be compiled into a `BINARY_OR` instruction by the compiler:

```
static int
binop(struct compiler *c, operator_ty op)
{
    switch (op) {
    case Add:
        return BINARY_ADD;
    ...
    case BitOr:
        return BINARY_OR;
```

In the evaluation loop, the case for a `BINARY_OR` will take two values from the value stack, the left, and right operation, then call `PyNumber_Or` against those 2 objects:

```
...
case TARGET(BINARY_OR): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *res = PyNumber_Or(left, right);
    Py_DECREF(left);
    Py_DECREF(right);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

The result, `res`, is then set as the top of the stack, overriding the current top value.

Value Stack Simulations

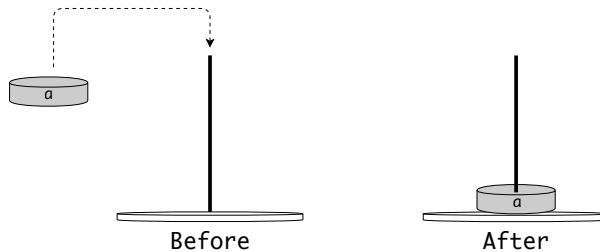
To understand the evaluation loop, you have to understand the value stack.

One way to think of the value stack is like a wooden peg on which you can stack cylinders. You would only add or remove one item at a time. This is done using the `PUSH(a)` macro, where `a` is a pointer to a `PyObject`.

For example, if you created a `PyLong` with the value `10` and pushed it onto the value stack:

```
PyObject *a = PyLong_FromLong(10);
PUSH(a);
```

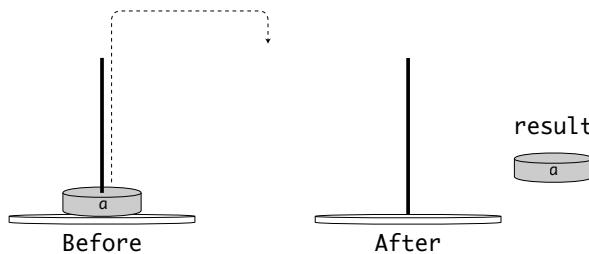
This action would have the following effect:



In the next operation, to fetch that value, you would use the `POP()` macro to take the top value from the stack:

```
PyObject *a = POP(); // a is PyLongObject with a value of 10
```

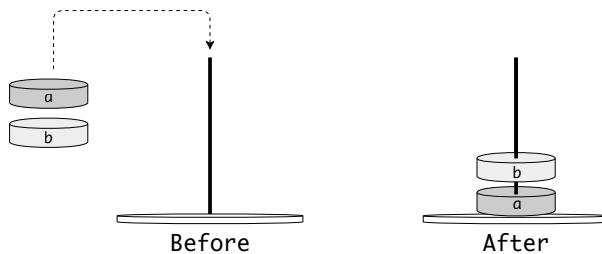
This action would return the top value and end up with an empty value stack:



If you were to add 2 values to the stack:

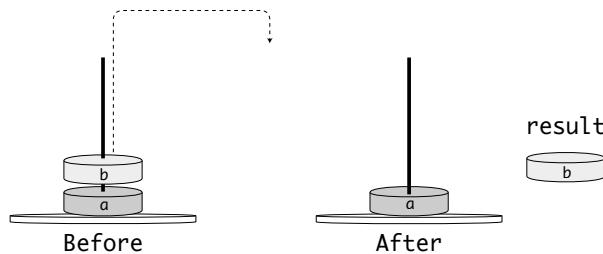
```
PyObject *a = PyLong_FromLong(10);
PyObject *b = PyLong_FromLong(20);
PUSH(a);
PUSH(b);
```

They would end up in the order in which they were added, so `a` would be pushed to the second position in the stack:



If you were to fetch the top value in the stack, you would get a pointer to `b` because it is at the top:

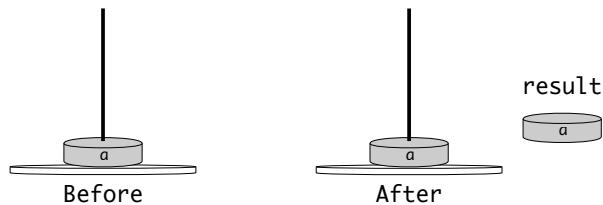
```
PyObject *val = POP(); // returns ptr to b
```



If you need to fetch the pointer to the top value in the stack without popping it, you can use the `PEEK(v)` operation, where `v` is the stack position:

```
PyObject *first = PEEK(0);
```

`0` represents the top of the stack, `1` would be the second position:



The `DUP_TWO()` macro can be used to clone the value at the top of the stack:

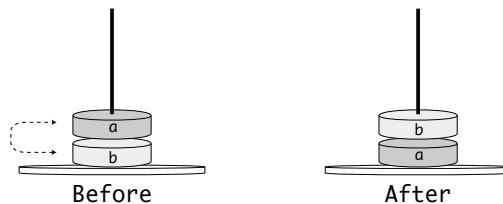
```
DUP_TOP();
```

This action would copy the value at the top to form 2 pointers to the same object:



There is a rotation macro `ROT_TWO` that swaps the first and second values.

```
ROT_TWO();
```



Stack Effects

Each of the opcodes has a predefined **stack effect**, calculated by the `stack_effect()` function inside `Python/compile.c`. This function returns the delta in the number of values inside the stack for each opcode. Stack effects can have a positive, negative, or zero value. Once the operation has been executed, if the stack effect (e.g., `+1`) does not match the delta in the value stack, an exception is raised to ensure there is no corruption to objects on the value stack.

Example: Adding an Item to a List

In Python, when you create a list, the `.append()` method is available on the list object:

```
my_list = []
my_list.append(obj)
```

In this example, `obj` is an object that you want to append to the end of the list.

There are 2 operations involved in this operation:

- `LOAD_FAST`, to load the object `obj` to the top of the value stack from the list of `locals` in the frame
- `LIST_APPEND` to add the object

First exploring `LOAD_FAST`, there are 5 steps:

1. The pointer to `obj` is loaded from `GETLOCAL()`, where the variable to load is the operation argument. The list of variable pointers is stored in `fastlocals`, which is a copy of the `PyFrame` attribute `f_localsplus`. The operation argument is a number, pointing to the index in the `fastlocals` array pointer. This means that the loading of a local is simply a copy of the pointer instead of having to look up the variable name.
2. If the variable no longer exists, an unbound local variable error is raised.
3. The reference counter for `value` (in our case, `obj`) is increased by 1.
4. The pointer to `obj` is pushed to the top of the value stack.
5. The `FAST_DISPATCH` macro is called, if tracing is enabled, the loop goes over again (with all the tracing). If tracing is not enabled, a `goto` is called to `fast_next_opcode`. The `goto` jumps back to the top of the loop for the next instruction.

```

...
    case TARGET(LOAD_FAST): {
        PyObject *value = GETLOCAL(oparg);           // 1.
        if (value == NULL) {
            format_exc_check_arg(
                PyExc_UnboundLocalError,
                UNBOUNDLOCAL_ERROR_MSG,
                PyTuple_GetItem(co->co_varnames, oparg));
            goto error;                            // 2.
        }
        Py_INCREF(value);                         // 3.
        PUSH(value);                            // 4.
        FAST_DISPATCH();                         // 5.
    }
...

```

The pointer to `obj` is now at the top of the value stack, and the next instruction, `LIST_APPEND`, is executed.

Many of the bytecode operations are referencing the base types, like `PyUnicode`, `PyNumber`. For example, `LIST_APPEND` appends an object to the end of a list. To achieve this, it pops the pointer from the value stack and returns the pointer to the last object in the stack. The macro is a shortcut for:

```
PyObject *v = (*--stack_pointer);
```

Now the pointer to `obj` is stored as `v`. The list pointer is loaded from `PEEK(oparg)`.

Then the C API for Python lists is called for `list` and `v`. The code for this is inside `Objects/listobject.c`, which you go into in the chapter Objects and Types.

A call to `PREDICT` is made, which guesses that the next operation will be `JUMP_ABSOLUTE`. The `PREDICT` macro has compiler-generated `goto` statements for each of the potential operations' `case` statements. This means the CPU can jump to that instruction and not have to go through the loop again:

```
...
case TARGET(LIST_APPEND): {
    PyObject *v = POP();
    PyObject *list = PEEK(oparg);
    int err;
    err = PyList_Append(list, v);
    Py_DECREF(v);
    if (err != 0)
        goto error;
    PREDICT(JUMP_ABSOLUTE);
    DISPATCH();
}
...
...
```

Note

Opcode Predictions

Some opcodes come in pairs, making it possible to predict the second code when the first is run. For example, `COMPARE_OP` is often followed by `POP_JUMP_IF_FALSE` or `POP_JUMP_IF_TRUE`.

“Verifying the prediction costs a single high-speed test of a register variable against a constant. If the pairing was good, then the processor’s own internal branch prediction has a high likelihood of success, resulting in a nearly zero-overhead transition to the next opcode. A successful prediction saves a trip through the eval-loop, including its unpredictable switch-case branch. Combined with the processor’s internal branch prediction, a successful `PREDICT` has the effect of making the two opcodes run as if they were a single new opcode with the bodies combined.”

If collecting opcode statistics, you have two choices:

1. Keep the predictions turned-on and interpret the results as if some opcodes had been combined
2. Turn off predictions so that the opcode frequency counter updates for both opcodes

Opcode prediction is disabled with threaded code since the latter allows the CPU to record separate branch prediction information for each opcode.

Some of the operations, such as `CALL_FUNCTION` and `CALL_METHOD`, have an operation argument referencing another compiled function. In these cases, another frame is pushed to the frame stack in the thread. The evaluation loop is then run for that function until the function completes. Each time a new frame is created and pushed onto the stack, the value of the frame’s `f_back` is set to the current frame before the new one is created.

This nesting of frames is clear when you see a stack trace:

```
cpython-book-samples▶ 31▶ example_stack.py
```

```
def function2():
    raise RuntimeError

def function1():
    function2()

if __name__ == '__main__':
    function1()
```

Calling this on the command-line will give you:

```
$ ./python example_stack.py
```

```
Traceback (most recent call last):
  File "example_stack.py", line 8, in <module>
    function1()
  File "example_stack.py", line 5, in function1
    function2()
  File "example_stack.py", line 2, in function2
    raise RuntimeError
RuntimeError
```

In `Lib▶ traceback.py`, the `walk_stack()` function can be used to get trace backs:

```
def walk_stack(f):
    """Walk a stack yielding the frame and line number for each frame.

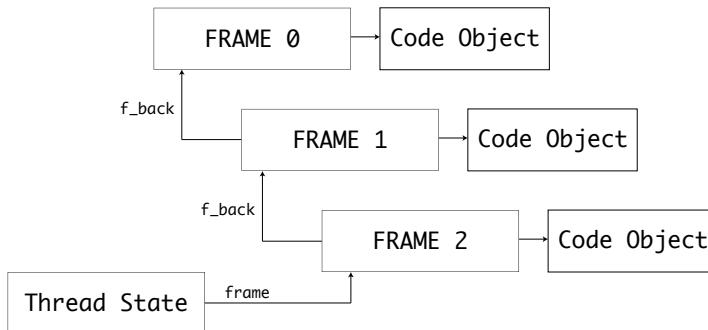
    This will follow f.f_back from the given frame. If no frame is given, the
    current stack is used. Usually used with StackSummary.extract.
    """
    if f is None:
        f = sys._getframe().f_back.f_back
    while f is not None:
        yield f, f.f_lineno
```

`f = f.f_back`

The parent's parent (`sys._getframe().f_back.f_back`) is set as the frame, because you don't want to see the call to `walk_stack()` or `print_trace()` in the traceback. The `f_back` pointer is followed to the top of the call stack.

`sys._getframe()` is the Python API to get the `frame` attribute of the current thread.

Here is how that frame stack would look visually, with 3 frames each with its code object and a thread state pointing to the current frame:



Conclusion

In this chapter, you've been introduced to the “brain” of CPython. The core evaluation loop is the interface between compiled Python code and the underlying C extension modules, libraries, and system calls.

Some parts in this chapter have been glossed over as you'll go into them in upcoming chapters. For example, the CPython interpreter has a core evaluation loop, you can have multiple loops running at the same time. Whether that be in parallel or concurrently. CPython can have multiple evaluation loops running multiple frames on a system. In an upcoming chapter on Parallelism and Concurrency, you will see

how the frame stack system is used for CPython to run on multiple core or CPUs. Also, CPython’s frame object API enables frames to be paused and resumed in the form of asynchronous programming.

The loading of variables using a Value Stack needs memory allocation and management. For CPython to run effectively, it has to have a solid Memory Management process. In the next chapter, you’ll explore that memory management process, and how it relates to the PyObject pointers used by the evaluation loop.

[Leave feedback on this section »](#)

Memory Management

The two most important parts of your computer are the memory and the CPU. They cannot work without the other, they must be utilized well, and they must be performant.

When designing a programming language, the authors need to make a vital trade-off:-

“How should the user manage computer memory?”

There are many solutions to this question. It depends on how simple you want the interface to be, whether you want the language to be cross-platform, whether you value performance over stability.

The authors of Python have made those decisions for you and left you with some additional ones to make yourself. In this chapter, you will explore how C manages memory, since CPython is written in C. You'll look at two critical aspects to managing memory in Python - **reference counting** and **garbage collection**.

By the end of this chapter, you will understand how CPython allocates memory on the Operating System, how object memory is allocated and freed, and how CPython manages memory leaks.

Memory Allocation in C

In C, variables must have their memory allocated from the Operating System before they can be used.

There are three memory allocation mechanisms in C:

1. Static memory allocation, where memory requirements are calculated at compile time and allocated by the executable when it starts
2. Automatic memory allocation, where memory requirements for a scope (e.g., function) are allocated within the call stack when a frame is entered and freed once the frame is terminated
3. Dynamic memory allocation, where memory can be requested and allocated dynamically (i.e., at runtime) by calls to the memory allocation API

Static Memory Allocation in C

Types in C have a fixed size. For global and static variables, the compiler will calculate the memory requirements for all global/static variables and compile that requirement into the application.

For example:

```
static int number = 0;
```

You can see the size of a type in C by using the `sizeof()` function. On my system, a 64-bit macOS running GCC, an `int` is 4 bytes. Basic types in C can have different sizes depending on the architecture and compiler.

For cases of arrays, these are statically defined, for example, an array of 10 integers:

```
static int numbers[10] = {0,1,2,3,4,5,6,7,8,9};
```

The C compiler converts this statement into an allocation of `sizeof(int) * 10` bytes of memory.

The C compiler uses system calls to allocate memory. These system calls depend on the Operating System and are low-level functions to the Kernel to allocate memory from the system memory pages.

Automatic Memory Allocation in C

Similar to static memory allocation, automatically memory allocation will calculate the memory allocation requirements at compile-time.

This example application will calculate 100 degrees Fahrenheit in Celsius:

```
cpython-book-samples▶32▶automatic.c
```

```
#include <stdio.h>

static const double five_ninths = 5.0/9.0;

double celsius(double fahrenheit){
    double c = (fahrenheit - 32) * five_ninths;
    return c;
}

int main() {
    double f = 100;
    printf("%f F is %f Cn", f, celsius(f));
    return 0;
}
```

Both static and dynamic memory allocation techniques are being used in the last example:

- The const value `five_ninths` is **statically** allocated because it has the `static` keyword
- The variable `c` within the function `celsius()` is allocated **automatically** when `celsius()` is called and freed when it is completed
- The variable `f` within the function `main()` is allocated **automatically** when `main()` is called and freed when it is completed
- The result of `celsius(f)` is implicitly allocated **automatically**
- The automatic memory requirements of `main()` are freed when the function completes

Dynamic Memory Allocation in C

In many cases, neither static nor automatically memory allocation is sufficient. For example- a program cannot calculate the size of the memory required at compile-time because it is a user-input defined.

In such cases, memory is allocated **dynamically**. Dynamic memory allocation works by calls to the C memory allocation APIs.

Operating Systems reserve a section of the system memory for dynamically allocation to processes. This section of memory is called a **heap**.

In this example, you will allocate memory dynamically to an array of Fahrenheit and celsius values.

The application takes some Fahrenheit values to print their celsius values:

```
cpython-book-samples▶ 32▶ dynamic.c
```

```
#include <stdio.h>
#include <stdlib.h>

static const double five_ninths = 5.0/9.0;

double celsius(double fahrenheit){
    double c = (fahrenheit - 32) * five_ninths;
    return c;
}

int main(int argc, char** argv) {
    if (argc != 2)
        return -1;
    int number = atoi(argv[1]);
    double* c_values = (double*)calloc(number, sizeof(double));
    double* f_values = (double*)calloc(number, sizeof(double));
    for (int i = 0 ; i < number ; i++ ){
        f_values[i] = (i + 10) * 10.0 ;
        c_values[i] = celsius((double)f_values[i]);
    }
}
```

```
    }
    for (int i = 0 ; i < number ; i++){
        printf("%f F is %f Cn", f_values[i], c_values[i]);
    }
    free(c_values);
    free(f_values);

    return 0;
}
```

If you execute this with the argument 4, it will print:

```
100.000000 F is 37.777778 C
110.000000 F is 43.333334 C
120.000000 F is 48.888888 C
130.000000 F is 54.444444 C
```

This example uses dynamic memory allocation to allocate a block of memory from the heap and then free's the memory back to the heap upon completion.

If any memory that is dynamically allocated is not freed, it will cause a memory leak.

Design of the Python Memory Management System

Being built on top of C, CPython has to use the constraints of **static**, **automatic**, and **dynamic** memory allocation.

There are some design aspects of the Python language that make those constraints even more challenging:

1. Python is a dynamically typed language. The size of variables cannot be calculated at compile-time
2. Most of Python's core types are dynamically sized. The `list` type can be of any size, `dict` can have any number of keys, even `int` is

dynamic. The user never has to specify the size of these types

3. Names in Python can be reused to values of different types, e.g.:

```
>>> a_value = 1
>>> a_value = "Now I'm a string"
>>> a_value = ["Now" , "I'm" "a", "list"]
```

To overcome these constraints, CPython relies heavily on dynamic memory allocation but adds safety-rails to automate the freeing of memory using the garbage collection and reference counting algorithms.

Instead of the Python developer having to allocate memory, Python Object memory is allocated automatically via a single, unified API.

This design requires that the entire CPython standard library and core modules (written in C) use this API.

Allocation Domains

CPython comes with three dynamic memory allocation domains:

1. **Raw Domain** - Used for allocation from the system heap. Used for large, or non-object related memory allocation
2. **Object Domain** - Used for allocation of all Python Object-related memory allocation
3. **PyMem Domain** - The same as PYMEM_DOMAIN_OBJ, exists for legacy API purposes

Each domain implements the same interface of functions:

- `_Alloc(size_t size)` - allocates memory of size, `size`, and returns a pointer
- `_Calloc(size_t nelem, size_t elsize)` - allocates memory of size
- `_Realloc(void *ptr, size_t new_size)` - reallocates memory to a new size

- `_Free(void *ptr)` - frees memory at `ptr` back to the heap

The `PyMemAllocatorDomain` enumeration represents the three domains in CPython as `PYMEM_DOMAIN_RAW`, `PYMEM_DOMAIN_OBJ`, and `PYMEM_DOMAIN_MEM`.

Memory Allocators

CPython uses two memory allocators:

1. The Operating System allocator (`malloc`) for the **Raw** memory domain
2. The CPython allocator (`pymalloc`) for the **PyMem** and **Object Memory** domains

Note

The CPython allocator, `pymalloc`, is compiled into CPython by default. You can remove it by recompiling CPython after setting `WITH_PYMalloc = 0` in `pyconfig.h`. If you remove it, the PyMem and Object memory domain APIs will use the system allocator.

If you compiled CPython with debugging (`--with-pydebug` on macOS/Linux, or with the `Debug` target on Windows), then each of the memory allocation functions will go to a Debug implementation. For example, with debugging enabled, your memory allocation calls would execute `_PyMem_DebugAlloc()` instead of `_PyMem_Alloc()`.

The CPython Memory Allocator

The CPython memory allocator sits atop the system memory allocator and has its algorithm for allocation. This algorithm is similar to the system allocator, except that it is customized to CPython:

- Most of the memory allocation requests are small, fixed-size because `PyObject` is 16 bytes, `PyASCIIObject` is 42 bytes, `PyCompactUnicodeObject` is 72 bytes, and `PyLongObject` is 32 bytes.

- The pymalloc allocator only allocates memory blocks up to 256KB, anything larger is sent to the system allocator
- The pymalloc allocator uses the GIL instead of the system thread-safety check

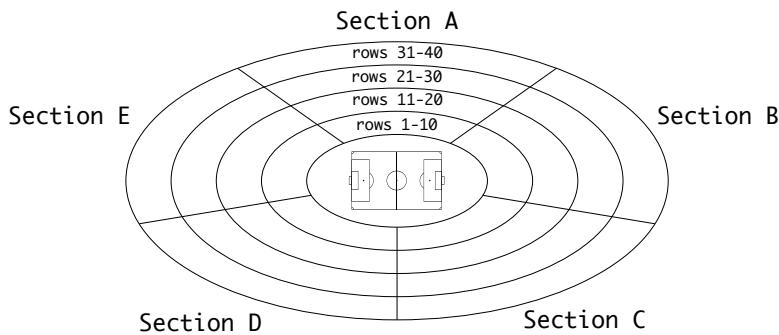
To help explain this situation, we're going to use a physical sports stadium as our analogy.

This is the stadium of “CPython FC,” our fictional team.

To help manage crowds, CPython FC has implemented a system of breaking the stadium up into sections A-E, each with seating rows 1-40.

The rows at the front of the stadium (1-10) are the Premium seats, each taking up more space. So there can only be 80 seats per row.

At the back, from rows 31-40 are the Economy seats. There are 150 seats per row:



- Just like the stadium has seats, the pymalloc algorithm has memory “**blocks**”

- Just like seats can either be Premium, Regular or Economy, blocks are all of a range of fixed sizes. You can't bring your deckchair
- Just like seats of the same size are put into rows, blocks of the same size are put into sized pools
- A central register keeps a record of where blocks are and the number of blocks available in a pool, just as the stadium would allocate seating
- When a row in the stadium is full, the next row is used. When a pool of blocks is full, the next pool is used
- Pools are grouped into arenas, just like the stadium groups the rows into sections

There are some advantages to this strategy:

1. The algorithm is more performant for CPython's main use case—short-lived, small objects
2. The algorithm uses the GIL instead of system thread lock detection
3. The algorithm uses memory mapping (mmap) instead of heap allocation

Related Source Files

Source files related to the memory allocator are:

File	Purpose
Include \blacktriangleright pymem.h	PyMem Allocator API
Include \blacktriangleright cpython \blacktriangleright pymem.h	PyMem Memory Allocator Configuration API
Include \blacktriangleright internal \blacktriangleright pycore_mem.h	GC data structure and internal APIs
Objects \blacktriangleright obmalloc.c	Domain allocator implementations, and the pymalloc implementation

Important Terms

- Requested memory is matched to a **block** size

- Blocks of the same size are all put into the same **pool** of memory
- Pools are grouped into **arenas**

Blocks, Pools, and Arenas

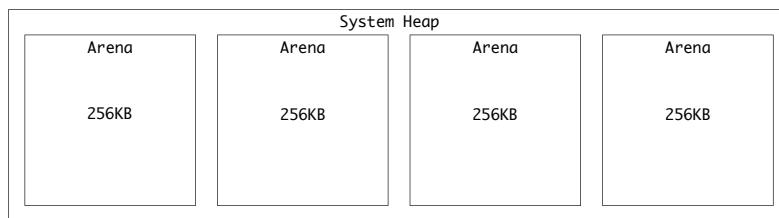
The largest group of memory is an arena. CPython creates arenas of 256KB to align with the system page size. A system page boundary is a fixed-length contiguous chunk of memory.

Even with modern, high-speed memory, contiguous memory will load faster than fragmented. It is beneficial to have contiguous memory.

Arenas

Arenas are allocated against the system heap, and with `mmap()` on systems supporting anonymous memory mappings. Memory mapping helps reduce heap fragmentation of the arenas.

This is the representation of 4 arenas within the system heap:



Arenas have the data struct `arenaobject`:

Field	Type	Purpose
<code>address</code>	<code>uintptr_t</code>	Memory address of the arena
<code>pool_address</code>	<code>block *</code>	Pointer to the next pool to be carved off for allocation
<code>nfreepools</code>	<code>uint</code>	The number of available pools in the arena: free pools + never-allocated pools
<code>ntotalpools</code>	<code>uint</code>	The total number of pools in the arena, whether or not available

Field	Type	Purpose
freepools	pool_header*	Singly-linked list of available pools
nextarena	arena_- object*	Next arena (see note)
prevarena	arena_- object*	Previous arena (see note)

Note

Arenas are linked together in a doubly-linked list inside the arena data structure, using the `nextarena` and `prevarena` pointers.

If this arena is **unallocated**, the `nextarena` member is used. The `nextarena` member links all unassociated arenas in the singly-linked `unused_arena_objects` global variable.

When this arena is associated with an allocated arena, with at least one available pool, both `nextarena` and `prevarena` are used in the doubly-linked `usable_arenas` list. This list is maintained in increasing order of `nfreepools` values.

Pools

Within an arena, pools are created for block sizes up to 512 bytes.

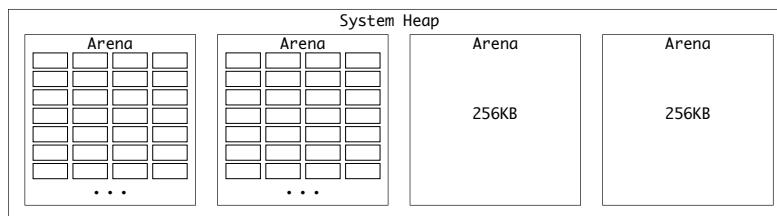
For 32-bit systems, the step is 8 bytes, so there are 64 classes:

Request in bytes	Size of allocated block	Size class index
1-8	8	0
9-16	16	1
17-24	24	2
25-32	32	3
...
497-504	504	62
505-512	512	63

For 64-bit systems, the step is 16 bytes, so there are 32 classes:

Request in bytes	Size of allocated block	Size class index
1-16	16	0
17-32	32	1
33-48	48	2
49-64	64	3
...
480-496	496	30
496-512	512	31

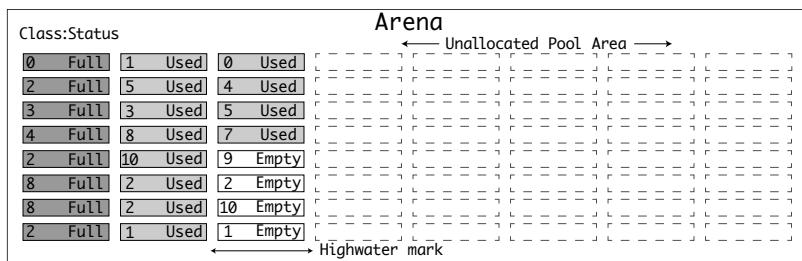
Pools are all 4096 bytes (4KB), so there are always 64 pools in an arena.



Pools are allocated on demand. When no available pools are available for the requested class size index, a new one is provisioned. Arenas have a “high water mark” to index how many pools have been provisioned.

Pools have three possible states,

1. **Full**: all available blocks in that pool are allocated
2. **Used**: the pool is allocated, and some blocks have been set, but it still has space
3. **Empty**: the pool is allocated, but no blocks have been set



Pools have the data structure `poolp`, which is a static allocation of the struct `pool_header`. The `pool_header` type has the following properties:

Field	Type	Purpose
<code>ref</code>	<code>uint</code>	Number of currently allocated blocks in this pool
<code>freeblock</code>	<code>block *</code>	Pointer to this pool’s “free list” head
<code>nextpool</code>	<code>pool_header*</code>	Pointer to the next pool of this size class
<code>prevpool</code>	<code>pool_header*</code>	Pointer to the previous pool of this size class
<code>arenaindex</code>	<code>uint</code>	Singly-linked list of available pools
<code>szidx</code>	<code>uint</code>	Class size index of this pool
<code>nextoffset</code>	<code>uint</code>	Number of bytes to unused block
<code>maxnextoffset</code>	<code>uint</code>	Maximum number that <code>nextoffset</code> can be until pool is full

Each pool of a certain class size will keep a doubly-linked list to the

next and previous pools of that class. When the allocation task happens, it is easy to jump between pools of the same class size within an arena by following this list.

Pool Tables

A register of the pools within an arena is called a **pool table**.

A pool table is a headed, circular, doubly-linked list of partially-used pools. The pool table is segmented by class size index, i . For an index of i , `usedpools[i + i]` points to the header of a list of all partially-used pools that have the size index for that class size.

Pool tables have some essential characteristics:

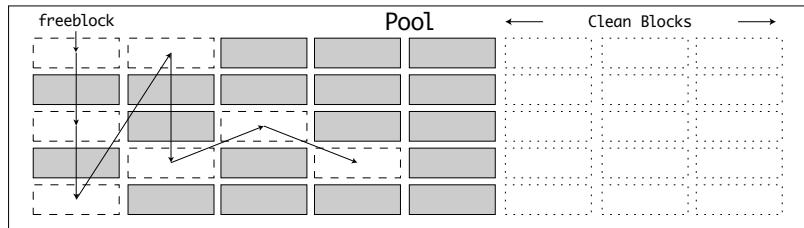
- When a pool becomes full, it is unlinked from its `usedpools[]` list.
- If a full block has a block freed, the pool back is put back in the used state. The newly freed pool is linked in at the front of the appropriate `usedpools[]` list so that the next allocation for its size class will reuse the freed block.
- On transition to empty, a pool is unlinked from its `usedpools[]` list, and linked to the front of its arena's singly-linked `freepools` list.

Blocks

Within a pool, memory is allocated into blocks. Blocks have the following characteristics:

- Within a pool, blocks of fixed class size can be allocated and freed.
- Available blocks within a pool are listed in the singly-linked list, `freeblock`.
- When a block is freed, it is inserted at the front of the `freeblock` list.
- When a pool is initialized, only the first two blocks are linked within the `freeblock` list.
- As long a pool is in the **used** state, there will be a block available for allocating.

A partially allocated pool with a combination of used, freed, and available blocks:



Block Allocation API

When a block of memory is requested by a memory domain that uses `pymalloc`, the `pymalloc_alloc` function will be called.

This function is a good place to insert a breakpoint and step through the code to test your knowledge of the blocks, pools, and arenas.

Objects → obmalloc.c line 1590

```
static inline void*
pymalloc_alloc(void *ctx, size_t nbytes)
{
    ...
}
```

For a request of `nbytes = 30`, it is neither zero, nor below the `SMALL_REQUEST_THRESHOLD` of `512`:

```
if (UNLIKELY(nbytes == 0)) {
    return NULL;
}
if (UNLIKELY(nbytes > SMALL_REQUEST_THRESHOLD)) {
    return NULL;
}
```

For a 64-bit system, the size class index is calculated as 1. This correlates to the 2nd class size index (17-32 bytes). The target pool is then `usedpools[1 + 1]` (`usedpools[2]`):

```

    uint size = (uint)(nbytes - 1) >> ALIGNMENT_SHIFT;
    poolp pool = usedpools[size + size];
    block *bp;

```

Next, a check is done to see if there is an available ('used') pool for the class size index. If the freeblock list is at the end of the pool, then there are still clean blocks available in that pool. `pymalloc_pool_extend()` is called to extend the freeblock list:

```

if (LIKELY(pool != pool->nextpool)) {
    /*
     * There is a used pool for this size class.
     * Pick up the head block of its free list.
     */
    ++pool->ref.count;
    bp = pool->freeblock;
    assert(bp != NULL);

    if (UNLIKELY((pool->freeblock = *(block **)bp) == NULL)) {
        // Reached the end of the free list, try to extend it.
        pymalloc_pool_extend(pool, size);
    }
}

```

If there were no available pools, a new pool is created and the first block is returned. The `allocate_from_new_pool()` function adds the new pool to the `usedpools` list automatically:

```

else {
    /*
     * There isn't a pool of the right size class immediately
     * available: use a free pool.
     */
    bp = allocate_from_new_pool(size);
}

return (void *)bp;
}

```

Finally, the new block address is returned.

Using the Python Debug API

The `sys` module contains an internal function, `_debugmallocstats()`, to get the number of blocks in use for each of the class size pools. It will also print the number of arenas allocated, reclaimed, and the total number of blocks used.

You can use this function to see the running memory usage:

```
$ ./python -c "import sys; sys._debugmallocstats()"

Small block threshold = 512, in 32 size classes.

class      size    num pools   blocks in use   avail blocks
-----      ---    -----      -----      -----
  0        16            1            181            72
  1        32            6            675            81
  2        48           18           1441            71
...
  2 free 18-sized PyTupleObjects * 168 bytes each =      336
  3 free 19-sized PyTupleObjects * 176 bytes each =      528
```

The output shows the class index size table, the allocations, and some additional statistics.

The Object and PyMem Memory Allocation Domains

CPython's object memory allocator is the first of the three domains that you will explore.

The purpose of the Object memory allocator is to allocate memory related to Python Objects, such as:

- New Object Headers
- Object data, such as dictionary keys and values, or list items

The allocator is also used for the compiler, AST, parser and evaluation

loop.

An excellent example of the Object memory allocator being used is the `PyLongObject` (`int`) type constructor, `PyLong_New()`.

- When a new `int` is constructed, memory is allocated from the Object Allocator.
- The size of the request is the size of the `PyLongObject` struct, plus the amount of memory required to store the digits.

Python longs are not equivalent to C's `long` type. They are a *list* of digits `[1,2,3,7,8,5,6,2,8,3,4]`.

The number `12378562834` in Python would be represented as the list of digits `[1,2,3,7,8,5,6,2,8,3,4]`.

This memory structure is how Python can deal with huge numbers without having to worry about 32 or 64-bit integer constraints.

Take a look at the `PyLong` constructor to see an example of object memory allocation:

```
PyLongObject *
_PyLong_New(Py_ssize_t size)
{
    PyLongObject *result;
    ...
    if (size > (Py_ssize_t)MAX_LONG_DIGITS) {
        PyErr_SetString(PyExc_OverflowError,
                       "too many digits in integer");
        return NULL;
    }
    result = PyObject_MALLOC(offsetof(PyLongObject, ob_digit) +
                           size*sizeof(digit));
    if (!result) {
        PyErr_NoMemory();
        return NULL;
    }
```

```
    return (PyLongObject*)PyObject_INIT_VAR(result, &PyLong_Type, size);  
}
```

If you were to call `_PyLong_New(2)`, it would calculate:

Value	Bytes
<code>sizeof(digit)</code>	4
<code>size</code>	2
<code>header offset</code>	26
Total	32

A call to `PyObject_MALLOC()` would be made with a `size_t` value of 32.

On my system, the maximum number of digits in a long, `MAX_LONG_DIGITS`, is 2305843009213693945 (a very, very big number). If you ran `_PyLong_New(2305843009213693945)` it would call `PyObject_MALLOC()` with a `size_t` of 9223372036854775804 bytes, or 8,589,934.592 Gigabytes (more RAM than I have available).

Using the `tracemalloc` Module

The `tracemalloc` in the standard library can be used to debug memory allocation through the Object Allocator. It provides information on where an object was allocated, and the number of memory blocks allocated.

As a debug tool, it is beneficial to calculate the amount of memory consumed by running your code or detect memory leaks.

To enable memory tracing, you should start Python with the `-x tracemalloc=1`, where 1 is the number of frames deep you want to trace. Alternatively, you can enable memory tracing using the `PYTHONTRACEMALLOC=1` environment variable. 1 is the number of frames deep you want to trace and can be replaced with any integer.

You can use the `take_snapshot()` function to create a snapshot instance, then compare multiple snapshots using `compare_to()`.

Create an example `tracedemo.py` file to see this in action:

```
cpython-book-samples▶ 32▶ tracedemo.py
```

```
import tracemalloc

tracemalloc.start()

def to_celsius(fahrenheit, /, options=None):
    return (fahrenheit-32)*5/9

values = range(0, 100, 10)  # values 0, 10, 20, ... 90

for v in values:
    c = to_celsius(v)

after = tracemalloc.take_snapshot()

tracemalloc.stop()
after = after.filter_traces([tracemalloc.Filter(True, '**/tracedemo.py')])
stats = after.statistics('lineno')

for stat in stats:
    print(stat)
```

Executing this will print a list of the memory used by line, from highest to lowest:

```
$ ./python -X tracemalloc=2 tracedemo.py

/Users/.../tracedemo.py:5: size=712 B, count=2, average=356 B
/Users/.../tracedemo.py:13: size=512 B, count=1, average=512 B
/Users/.../tracedemo.py:11: size=480 B, count=1, average=480 B
/Users/.../tracedemo.py:8: size=112 B, count=2, average=56 B
/Users/.../tracedemo.py:6: size=24 B, count=1, average=24 B
```

The line with the highest memory consumption was `return (fahrenheit-32)*5/9` (the actual calculation).

The Raw Memory Allocation Domain

The Raw Memory Allocation domain is used either directly, or when the other two domains are called with a request size over 512 KB.

It takes the request size, in bytes, and calls `malloc(size)`.

If the size argument is 0, some systems will return NULL for `malloc(0)`, which would be treated as an error.

Some platforms would return a pointer with no memory behind it, which would break `pymalloc`. To solve these problems, `_PyMem_RawMalloc()` will add an extra byte before calling `malloc()`.

Important

By default, the PyMem Domain allocators will use the Object Allocators. `PyMem_Malloc()` and `PyObject_Malloc()` will have the same execution path.

Custom Domain Allocators

CPython also allows for the allocation implementation for any of the three domains to be overridden. If your system environment required bespoke memory checks or algorithms for memory allocation, then you can plug a new set of allocation functions into the runtime.

`PyMemAllocatorEx` is a `typedef struct` with members for all of the methods you would need to implement to override the allocator:

```
typedef struct {
    /* user context passed as the first argument to the 4 functions */
    void *ctx;

    /* allocate a memory block */
    void* (*malloc) (void *ctx, size_t size);

    /* allocate a memory block initialized by zeros */
    void* (*calloc) (void *ctx, size_t nmemb, size_t size);

    /* free a memory block */
    void (*free) (void *ctx, void *ptr);
}
```

```
void* (*calloc) (void *ctx, size_t nelem, size_t elsize);

/* allocate or resize a memory block */
void* (*realloc) (void *ctx, void *ptr, size_t new_size);

/* release a memory block */
void (*free) (void *ctx, void *ptr);
} PyMemAllocatorEx;
```

The API method `PyMem_GetAllocator()` is available to get the existing implementation:

```
PyMemAllocatorEx * existing_obj;
PyMem_GetAllocator(PYMEM_DOMAIN_OBJ, existing_obj);
```

Important

There are some important design tests for custom allocators:

- The new allocator must return a distinct non-NULL pointer when requesting zero bytes
- For the PYMEM_DOMAIN_RAW domain, the allocator must be thread-safe

If you implemented functions `My_Malloc`, `My_Calloc`, `My_Realloc` and `My_Free` implementing the signatures in `PyMemAllocatorEx`, you could override the allocator for any domain, e.g., the `PYMEM_DOMAIN_OBJ` domain:

```
PyMemAllocatorEx my_allocators =
{NULL, My_Malloc, My_Calloc, My_Realloc, My_Free};
PyMem_SetAllocator(PYMEM_DOMAIN_OBJ, &my_allocators);
```

Custom Memory Allocation Sanitizers

Memory allocation sanitizers are an additional algorithm placed between the system call to allocate memory, and the kernel function to allocate the memory on the system. They are used for environments

that require specific stability constraints, very high security, or for debugging memory allocation bugs.

C_Python can be compiled using several memory sanitizers. These are part of the compiler libraries, not something developed for C_Python.

They typically slow down C_Python significantly and cannot be combined. They generally are for use in debugging scenarios or systems where preventing corrupt memory access is critical.

Address Sanitizer

Address Sanitizer is a “fast” memory error detector. It can detect many runtime memory-related bugs:

- Out-of-bounds accesses to heap, stack, and globals
- Memory being used after it has been freed
- Double-free, invalid free

It can be enabled by running:

```
$ ./configure --with-address-sanitizer ...
```

Important

Address Sanitizer would slow down applications by up to 2x and consume up to 3x more memory.

Address Sanitizer is supported on:

- Linux
- macOS
- NetBSD
- FreeBSD

See the official documentation for more information.

Memory Sanitizer

Memory Sanitizer is a detector of uninitialized reads. If an address space is addressed before it has been initialized (allocated), then the process is stopped before the memory can be read.

It can be enabled by running:

```
$ ./configure --with-memory-sanitizer ...
```

Important

Memory Sanitizer would slow down applications by up to 2x and consume up to 2x more memory.

Memory Sanitizer is supported on:

- Linux
- NetBSD
- FreeBSD

[See the official documentation for more information.](#)

Undefined Behavior Sanitizer

Undefined Behavior Sanitizer is a “fast” undefined behavior detector. It can catch various kinds of undefined behavior during execution, for example:

- Using misaligned or null pointer
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination

It can be enabled by running:

```
$ ./configure --with-undefined-behavior-sanitizer ...
```

Undefined Behavior Sanitizer is supported on:

- Linux
- macOS
- NetBSD
- FreeBSD

See the [official documentation](#) for more information.

The Undefined Behavior Sanitizer has many configurations, using `--with-undefined-behavior-sanitizer` will set the `undefined` profile. To use another profile, e.g., `nullability`, run `./configure` with the custom `CFLAGS`:

```
$ ./configure CFLAGS="-fsanitize=nullability"  
LDFLAGS="-fsanitize=nullability"
```

The PyArena Memory Arena

Throughout this book, you will see references to a `PyArena` object.

The `PyArena` is a separate arena allocation API used for the compiler, frame evaluation, and other parts of the system not run from Python's Object allocation API. The `PyArena` also has its own list of allocated objects within the arena structure. Memory allocated by the `PyArena` is not a target of the garbage collector.

When memory is allocated in a `PyArena` instance, it will capture a running total of the number of blocks allocated, then call `PyMem_Alloc`.

Allocation requests to the `PyArena` use the Object Allocator for blocks $\leq 512\text{KB}$, or the Raw Allocator for blocks $> 256\text{KB}$.

Related Files

File	Purpose
Include ➔ <code>pyarena.h</code>	The PyArena API and type definitions
Python ➔ <code>pyarena.c</code>	The PyArena implementation

Reference Counting

As you have explored so far in this chapter, CPython is built on C's Dynamic Memory Allocation system. Memory requirements are determined at runtime, and memory is allocated on the system using the `PyMem` APIs.

For the Python developer, this system has been abstracted and simplified. Developers don't have to worry (too much) about allocating and free'ing memory.

To achieve simple memory management, Python adopts two strategies for managing the memory allocated by objects:

1. Reference Counting
2. Garbage Collection

Creating Variables in Python

To create a variable in Python, you have to assign a value to a *uniquely* named variable. For example:

```
my_variable = ['a', 'b', 'c']
```

When a **value** is assigned to a **variable** in Python, the **name** of the variable is checked within the `locals` and `globals` scope to see if it already exists.

In the example, `my_variable` is not already within any `locals()` or `globals()` dictionary. A new list object is created, and a pointer is stored in the `locals()` dictionary. There is now one **reference** to `my_variable`. The list object's memory should not be freed while there are valid references to it. If memory were freed, the `my_variable`

pointer would point to invalid memory space, and CPython would crash.

Throughout the C source code for CPython, you will see calls to `Py_INCREF()` and `Py_DECREF()`.

These macros are the primary API for incrementing and decrementing references to Python objects. Whenever something depends on a value, the reference count increases, when that dependency is no longer valid, the reference count decreases.

If a reference count reaches zero, it is assumed that it is no longer needed, and it is automatically freed.

Incrementing References

Every instance of `PyObject` has a property `ob_refcnt`. This property is a counter of the number of references to that object.

References to an object are incremented under many scenarios. In the CPython code base, there are over 3000 calls to `Py_INCREF()`. The most frequent calls are when an object is:

- assigned to a variable name
- referenced as a function or method argument
- returned, or yielded from a function

The logic behind the `Py_INCREF` macro is simple. It increments the `ob_refcnt` value by 1:

```
static inline void _Py_INCREF(PyObject *op)
{
    _Py_INC_REFTOTAL;
    op->ob_refcnt++;
}
```

If CPython is compiled in debug mode, `_Py_INC_REFTOTAL` will increment a global reference counter, `_Py_RefTotal`.

Note

You can see the global reference counter by adding the `-X showrefcount` flag when running CPython:

```
$ ./python -X showrefcount -c "x=1; x+=1; print(f'x is {x}')"
x is 2
[18497 refs, 6470 blocks]
```

The first number in brackets is the number of references made during the process, and the second is the number of allocated blocks.

Decrementing References

References to an object are decremented when a variable falls outside of the scope in which it was declared. Scope in Python can refer to a function or method, a comprehension, or a lambda. These are some of the more literal scopes, but there are many other implicit scopes, like passing variables to a function call.

The `Py_DECREF()` function is more complex than `Py_INCREF()` because it also handles the logic of a reference count reaching 0, requiring the object memory to be freed:

```
static inline void _Py_DECREF(
#ifndef Py_REF_DEBUG
    const char *filename, int lineno,
#endif
    PyObject *op)
{
    _Py_DEC_REFTOTAL;
    if (--op->ob_refcnt != 0) {
#ifndef Py_REF_DEBUG
        if (op->ob_refcnt < 0) {
            _Py_NegativeRefcount(filename, lineno, op);
        }
#endif
    }
}
```

```
    }
    else {
        _Py_Dealloc(op);
    }
}
```

Inside `Py_DECREF()`, when the reference counter (`ob_refcnt`) value becomes 0, the object destructor is called via `_Py_Dealloc(op)`, and any allocated memory is freed.

As with `Py_INCREF()`, there are some additional functions when CPython has been compiled in debug mode.

For an increment, there should be an equivalent decrement operation.

If a reference count becomes a negative number, this indicates an imbalance in the C code. An attempt to decrement references to an object that has no references will give this error message:

```
<file>:<line>: _Py_NegativeRefCount: Assertion failed:
    object has negative ref count
Enable tracemalloc to get the memory block allocation traceback

object address  : 0x109eaac50
object refcount : -1
object type     : 0x109cadf60
object type name: <type>
object repr     : <refcnt -1 at 0x109eaac50>
```

When making changes to the behavior of an operation, the Python language, or the compiler, you must carefully consider the impact on object references.

Reference Counting in Bytecode Operations

A large portion of the reference counting in the Python happens within the bytecode operations in `Python>ceval.c`.

Take this example, how many references do you think there are to `y`?

```
y = "hello"

def greet(message=y):
    print(message.capitalize() + " " + y)

messages = [y]

greet(*messages)
```

At a glance, `y` is immediately referenced by:

1. `y` is a variable in the top-level scope
2. `y` is referenced as a default value for the keyword argument `message`
3. `y` is referenced inside the `greet()` function
4. `y` is an item in the `messages` list

Run this code with an additional snippet:

```
import sys
print(sys.getrefcount(y))
```

The total references to `y` is 6.

Instead of the logic for incrementing and decrementing references sitting within a central function that has to cater for all these cases (and more!), the logic is split into small parts.

A bytecode operation should have a determining impact on the reference counter for the objects that it takes as arguments.

For example, in the frame evaluation loop, the `LOAD_FAST` operation loads the object with a given name and pushes it to the top of the value stack. Once the variable name, which is provided in the `oparg`, has been resolved using `GETLOCAL()` the reference counter is incremented:

```
case TARGET(LOAD_FAST): {
    PyObject *value = GETLOCAL(oparg);
```

```

if (value == NULL) {
    format_exc_check_arg(tstate, PyExc_UnboundLocalError,
                         UNBOUNDLOCAL_ERROR_MSG,
                         PyTuple_GetItem(co->co_varnames, oparg));
    goto error;
}
Py_INCREF(value);
PUSH(value);
FAST_DISPATCH();
}

```

A LOAD_FAST operation is compiled by many AST nodes that have operations.

For example, if you were to assign two variables `a` and `b`, then create third, `c` from the result of multiplying them:

```

a = 10
b = 20
c = a * b

```

In the third operation, `c = a * b`, the right-hand side expression, `a * b`, would be assembled into three operations:

1. LOAD_FAST, resolving the variable `a` and pushing it to the value stack then incrementing the references to `a` by 1
2. LOAD_FAST, resolving the variable `b` and pushing it to the value stack then incrementing the references to `b` by 1
3. BINARY_MULTIPLY

The binary multiply operator, `BINARY_MULTIPLY` knows that references to the left and right variables in the operation have been loaded to the first and second positions in the value stack. It is also implied that the `LOAD_FAST` operation incremented its reference counters.

In the implementation of the `BINARY_MULTIPLY` operation, the references to both `a` (left) and `b` (right) are decremented once the result has been calculated.

```
case TARGET(BINARY_MULTIPLY): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *res = PyNumber_Multiply(left, right);
    Py_DECREF(left);
    Py_DECREF(right);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

The resulting number, `res`, will have a reference count of 1 before it is set as the top of the value stack.

Conclusion

CPython's reference counter has the benefits of being simple, fast, and efficient.

The biggest drawback of the reference counter is that it needs to cater for, and carefully balance, the effect of every operation.

As you just saw, a bytecode operation increments the counter, and it is assumed that an equivalent operation will decrement it properly. What happens if there's an unexpected error? Have all possible scenarios been tested?

Everything discussed so far is within the realm of the CPython runtime. The Python developer has little to no control over this behavior.

There is also a significant flaw in the reference counting approach—**cyclical references**.

Take this Python example:

```
x = []
x.append(x)
del x
```

The reference count for `x` is still 1 because it referred to itself.

To cater to this complexity, and resolve some of these memory leaks, CPython has a second memory management mechanism, **Garbage Collection**.

Garbage Collection

How often does your garbage get collected? Weekly or fortnightly?

When you're finished with something, you discard it and throw it in the trash. But that trash won't get collected straight away. You need to wait for the garbage trucks to come and pick it up.

CPython has the same principle, using a garbage collection algorithm. CPython's garbage collector is **enabled** by default, happens in the background, and works to deallocate memory that's been used for objects which no longer exist.

Because the garbage collection algorithm is a lot more complicated than the reference counter, it doesn't happen all the time. If it did, it would consume a vast amount of CPU resources. The garbage collection runs periodically after a set number of operations.

Related Source Files

Source files related to the garbage collector are:

File	Purpose
Modules \blacktriangleright <code>gcmodule.c</code>	The Garbage Collection module and algorithm implementation
Include \blacktriangleright internal \blacktriangleright <code>pycore_mem.h</code>	The GC data structure and internal APIs

The GC Design

As you uncovered in the previous section, every Python object retains a counter of the number of references to it. Once that counter reaches

o, the object is finalized, and the memory is freed.

Many of the Python **container types**, like lists, tuples, dictionaries, and sets, could result in cyclical references. The reference counter is an insufficient mechanism to ensure that objects which are no longer required are freed. While creating cyclical references in containers should be avoided, there are many examples within the standard library and the core interpreter.

Here is another common example, where a container type (`class`) can refer to itself:

```
cpython-book-samples▶32▶user.py
```

```
__all__ = ['User']

class User(BaseUser):
    name: 'str' = ""
    login: 'str' = ""

    def __init__(self, name, login):
        self.name = name
        self.login = login
        super(User).__init__()

    def __repr__(self):
        return ""

class BaseUser:
    def __repr__(self):
        # This creates a cyclical reference
        return User.__repr__(self)
```

In this example, the instance of `User` links to the `BaseUser` type, which references back to the instance of `User`.

The goal of the garbage collector is to find **unreachable** objects and mark them as garbage.

Some GC algorithms, like mark-and-sweep, or stop-and-copy start at the root of the system and explore all *reachable* objects. This is hard to do in CPython because C extension modules can define and store their own objects. You could not easily determine all objects by simply looking at `locals()` and `globals()`.

For long-running processes, or large data processing tasks, running out of memory would cause a significant issue.

Instead, the CPython garbage collector leverages the existing reference counter and a custom garbage collector algorithm to find all *unreachable* objects. Because the reference counter is already in place, the role of the CPython garbage collector is to look for cyclical references in certain container types.

Container Types Included in GC

The Garbage Collector will look for types that have the flag `Py_TPFLAGS_HAVE_GC` set in their type definition.

You will cover type definitions in the chapter Objects in CPython.

Types that are marked for garbage collection are:

- Class, Method and Function objects
- Cell Objects
- Byte arrays, Byte, and Unicode strings
- Dictionaries
- Descriptor Objects, used in attributes
- Enumeration Objects
- Exceptions
- Frame Objects
- Lists, Tuples, Named Tuples and Sets
- Memory Objects
- Modules and Namespaces

- Type and Weak Reference Objects
- Iterators and Generators
- Pickle Buffers

Wondering what's missing? Floats, Integers, Boolean, and `NoneType` are not marked for garbage collection.

Custom types written with C extension models can be marked as requiring GC using the [GC C-API](#).

Untrackable Objects and Mutability

The GC will track certain types for changes in their properties to determine which are unreachable.

Some container instances are not subject to change because they are immutable, so the API provides a mechanism for “untracking.” The fewer objects there are to be tracked by the GC, the faster and more efficient the GC is.

An excellent example of untrackable objects is tuples. Tuples are immutable. Once you create them, they cannot be changed. However, tuples can contain mutable types, like lists and dictionaries.

This design in Python creates many side-effects, one of which is the GC algorithm. When a tuple is created, unless it is empty, it is marked for tracking. When the GC runs, every tuple looks at its contents to see if it only contains immutable (untracked) instances. This step is completed in [_PyTuple_MaybeUntrack\(\)](#). If the tuple determines that it only contains immutable types, like booleans and integers, it will remove itself from the GC tracking by calling [_PyObject_GC_UNTRACK\(\)](#).

Dictionaries are empty when they are created and untracked. When an item is added to a dictionary, if it is a tracked object, the dictionary will request itself to be tracked by the GC.

You can see if any object is being tracked by calling `gc.is_tracked(obj)`.

Garbage Collection Algorithm

See Also

The CPython core development team has written a [detailed guide](#) on the GC algorithm.

Initialization

The `PyGC_Collect()` entry-point follows five steps to start, and stop the garbage collector.

1. Get the Garbage Collection state, `GCState` from the interpreter
2. Check to see if the GC is enabled
3. Check to see if the GC is already running
4. Run the collection function, `collect()` with progress callbacks
5. Mark the GC as completed

When the collection stage is run and completed, callback methods can be user-specified by the `gc.callbacks` list. Callbacks should have a method signature `f(stage: str, info: dict):`

```
Python 3.9.0b1 (tags/v3.9.0b1:97fe9cf, May 19 2020, 10:00:00)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> import gc
>>> def gc_callback(phase, info):
...     print(f"GC phase:{phase} with info:{info}")
...
>>> gc.callbacks.append(gc_callback)
>>> x = []
>>> x.append(x)
>>> del x
>>> gc.collect()

GC phase:start with info:{'generation': 2, 'collected': 0, 'uncollectable': 0}
GC phase:stop with info:{'generation': 2, 'collected': 1, 'uncollectable': 0}
```

The Collection Stage

In the main GC function, `collect()` targets a particular generation. There are 3 generations in CPython. Before you understand the purpose of the generations, it's important to understand the collection algorithm.

For each collection, the GC will use a doubly-linked list of type `PyGC_HEAD`.

So that the GC doesn't have to "find" all container types, all container types that are a target for the GC have an additional header. This header links them all together in a doubly-linked list. When one of these container types is created, it adds itself to the list, and when it is destroyed, it removes itself.

You can see an example of this in the `cellobject.c` type:

Objects ▶ `cellobject.c` line 7

```
PyObject *
PyCell_New(PyObject *obj)
{
    PyCellObject *op;

    op = (PyCellObject *)PyObject_GC_New(PyCellObject, &PyCell_Type);
    if (op == NULL)
        return NULL;
    op->ob_ref = obj;
    Py_XINCREF(obj);

    >> _PyObject_GC_TRACK(op);
    return (PyObject *)op;
}
```

Because cells are mutable, the object is marked to be tracked by a call to `_PyObject_GC_TRACK()`.

When cell objects are deleted, the `cell_dealloc()` function is called. This function takes three steps:

1. The destructor tells the GC to stop tracking this instance by calling `_PyObject_GC_UNTRACK()`. Because it has been destroyed, it's contents don't need to be checked for changes in subsequent collections.
2. `Py_XDECREF` is a standard call in any destructor to decrement the reference counter. The reference counter for an object is initialized to 1, so this counters that operation.
3. The `PyObject_GC_Del()` will remove this object from the GC linked-list by calling `gc_list_remove()` and then free the memory with `PyObject_FREE()`.

Objects▶ `cellobjobject.c` line 79

```
static void
cell_dealloc(PyCellObject *op)
{
    _PyObject_GC_UNTRACK(op);
    Py_XDECREF(op->ob_ref);
    PyObject_GC_Del(op);
}
```

When a collection starts, it will merge younger generations into the current. For example, if you are collecting the second generation, when it starts collecting, it will merge the first generation's objects into the GC list using `gc_list_merge()`.

The GC will then determine unreachable objects in the young (currently targeted) generation.

The logic for determining unreachable objects is located in `deduce_unreachable()`. It follows these stages:

1. For every object in the generation, copy the reference count value `ob->ob_refcnt` to `ob->gc_ref`.
2. For every object, subtract internal (cyclical) references from `gc_refs` to determine how many objects can be collected by the GC. If `gc_refs` ends up equal to 0, that means it is unreachable.
3. Create a list of unreachable objects and add every object that met

- the criteria in (2) to it.
4. Remove every object that met the criteria in (2) from the generation list.

There is no single method for determining cyclical references. Each type must define a custom function with signature `traverseproc` in the `tp_traverse` slot. To complete task (2), the `deduce_unreachable()` function will call the traversal function for every object within a `subtract_refs()` function.

It is expected that the traversal function will run the callback `visit_decref()` for every item it contains:

Modules ▶ `gcmodule.c` line 462

```
static void
subtract_refs(PyGC_Head *containers)
{
    traverseproc traverse;
    PyGC_Head *gc = GC_NEXT(containers);
    for (; gc != containers; gc = GC_NEXT(gc)) {
        PyObject *op = FROM_GC(gc);
        traverse = Py_TYPE(op)->tp_traverse;
        (void) traverse(FROM_GC(gc),
                        (visitproc)visit_decref,
                        op);
    }
}
```

The traversal functions are kept within each object's source code in `Objects`. For example, the `tuple` type's traversal, `tupletraverse()` calls `visit_decref()` on all of it's items. The dictionary type, will call `visit_decref()` on all keys and values.

Any object which did not end up being moved to the `unreachable` list graduates to the next generation.

Freeing Objects

Once unreachable objects have been determined, they can be (carefully) freed following these stages.

The approach depends on whether the type implements the old or the new finalizer slot:

1. If an object has defined a finalizer in the legacy `tp_del` slot, it cannot safely be deleted and is marked as `uncollectable`. These are added to the `gc.garbage` list for the developer to destroy manually.
2. If an object has defined a finalizer in the `tp_finalize` slot, mark the objects as finalized to avoid calling them twice.
3. If an object in (2) has been “resurrected” by being initialized again, the GC reruns the collection cycle.
4. For all objects, call the `tp_clear` slot. This slot changes the reference count, `ob_refcnt`, to 0, triggering the freeing of memory.

Generational GC

Generational garbage collection is a technique based on the observation that most (80%+) objects are destroyed shortly after being created.

C_Python’s GC uses three generations that have thresholds to trigger their collections. The youngest generation (0) has a high threshold to avoid the collection loop being run too frequently. If an object survives the GC, it will move to the second generation, and then the third.

In the collection function, a single generation is targeted, and it merges younger generations into it before execution. For this reason, if you run `collect()` on generation 1, it will collect generation 0. Likewise, running `collect` on generation 2 will `collect()` generations 0 and 1.

When objects are instantiated, the generational counters are incremented. When the counter reaches a user-defined threshold,

`collect()` is automatically run.

Using the GC API From Python

CPython's standard library comes with a Python module to interface with the arena and the garbage collector, the `gc` module. Here's how to use the `gc` module in debug mode:

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
```

This will print the statistics whenever the garbage collector is run:

```
gc: collecting generation 2...
gc: objects in each generation: 3 0 4477
gc: objects in permanent generation: 0
gc: done, 0 unreachable, 0 uncollectable, 0.0008s elapsed
```

You use the `gc.DEBUG_COLLECTABLE` to discover when items are collected for garbage. When you combine this with the `gc.DEBUG_SAVEALL` debug flag, it will move items to a list, `gc.garbage` once they have been collected:

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_COLLECTABLE | gc.DEBUG_SAVEALL)
>>> z = [0, 1, 2, 3]
>>> z.append(z)
>>> del z
>>> gc.collect()
gc: collectable <list 0x10d594a00>
>>> gc.garbage
[[0, 1, 2, 3, [...]]]
```

You can get the threshold after which the garbage collector is run by calling `get_threshold()`:

```
>>> gc.get_threshold()
(700, 10, 10)
```

You can also get the current threshold counts:

```
>>> gc.get_count()  
(688, 1, 1)
```

Lastly, you can run the collection algorithm manually for a generation, and it will return the collected total:

```
>>> gc.collect(0)  
24
```

If you don't specify a generation, it will default to 2, which merges generations 0 and 1:

```
>>> gc.collect()  
20
```

Conclusion

In this chapter, you've been shown how CPython allocates, manages, and frees memory. These operations happen 1000s of times during the lifecycle of even the simplest Python script. The reliability and scalability of CPython's Memory Management system are what enables it to scale from a 2-line script all the way to run some of the world's biggest websites.

The Object and Raw Memory Allocation systems you've been shown in this chapter will come in useful if you develop C extension modules. C extension modules require an intimate knowledge of CPython's Memory Management system. Even a single missing `Py_INCREF()` can cause a memory leak or system crash.

When working with pure Python code, knowledge of the GC is useful if you're designing long-running Python code. For example, if you designed a single function that executes over hours, days, or even longer. This function would need to carefully manage its memory within the constraints of the system on which it's executing. You can use some of the techniques learned in this chapter to control and tweak the GC

generations to better optimize your code and its memory footprint.

[Leave feedback on this section »](#)

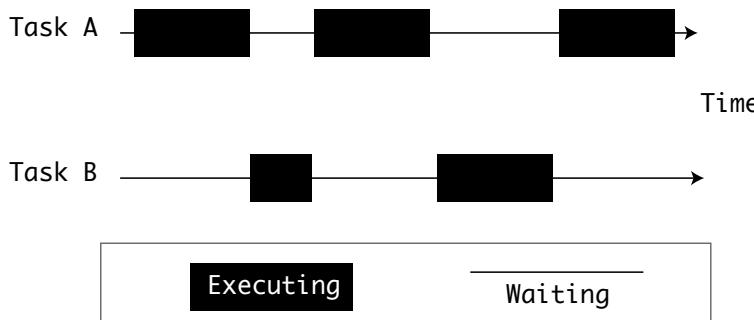
Parallelism and Concurrency

The first computers were designed to do one thing at a time. A lot of their work was in the field of computational mathematics. As time went on, computers are needed to process inputs from a variety of sources. Some input as far away as distant galaxies. The consequence of this is that computer applications spend a lot of time idly waiting for responses. Whether they be from a bus, an input, memory, computation, an API, or a remote resource.

Another progression in computing was the move in Operating Systems away from a single-user terminal, to a multitasking Operating System. Applications needed to run in the background to listen and respond on the network and process inputs such as the mouse cursor. Multitasking was required way before modern multiple-core CPUs, so Operating Systems long could to share the system resources between multiple processes.

At the core of any Operating System is a registry of running processes. Each process will have an owner, and it can request resources, like memory or CPU. In the last chapter, you explored memory allocation. For a CPU, the process will request CPU time in the form of operations to be executed. The Operating System controls which process is using the CPU. It does this by allocating “CPU Time” and scheduling processes by a priority:

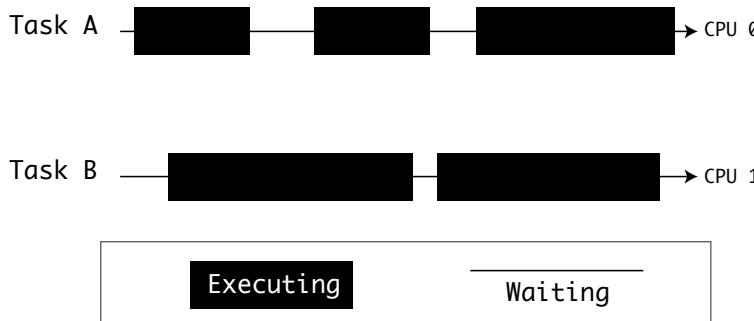
Concurrent Model



A single process may need to do multiple things at once. For example, if you use a word processor, it needs to check your spelling while you're typing. Modern applications accomplish this by running multiple threads, concurrently, and handling their own resources.

Concurrency is an excellent solution to dealing with multitasking, but CPUs have their limits. Some high-performance computers deploy either multiple CPUs or multiple cores to spread tasks. Operating Systems provide a way of scheduling processes across multiple CPUs:

Parallel Model



In summary,

- To have **parallelism**, you need multiple computational units. Computational units can be CPUs or Cores.
- To have **concurrency**, you need a way of scheduling tasks so that idle ones don't lock the resources.

Many parts of CPython's design abstract the complexity of Operating Systems to provide a simple API for developers. CPython's approach to parallelism and concurrency is no exception.

Models of Parallelism and Concurrency

CPython offers many approaches to Parallelism and Concurrency. Your choice depends on several factors. There are also overlapping use cases across models as CPython has evolved.

You may find that for a particular problem, there are two or more concurrency implementations to choose from. Each with their own pros and cons.

The four bundled models with CPython are:

Approach	Module	Concurrent	Parallel
Threading	<code>threading</code>	Yes	No
Multiprocessing	<code>multiprocessing</code>	Yes	Yes
Async	<code>asyncio</code>	Yes	No
Subinterpreters	<code>subinterpreters</code>	Yes	Yes

The Structure of a Process

One of the tasks for an Operating System, like Windows, macOS, or Linux, is to control the running processes. These processes could be UI applications like a browser or IDE. They could also be background processes, like network services or OS services.

To control these processes, the OS provides an API to start a new process. When a process is created, it is registered by the Operating System so that it knows which processes are running. Processes are given a unique ID (PID). Depending on the Operating System, they have other properties.

POSIX processes have a minimum set of properties, registered in the Operating System:

- Controlling Terminal
- Current Working Directory
- Effective Group ID, Effective User ID
- File Descriptors, File Mode Creation Mask
- Process Group ID, Process ID
- Real Group ID, Real User ID
- Root Directory

You can see these attributes for running processes in macOS or Linux by running the `ps` command.

See Also

The [IEEE POSIX Standard \(1003.1-2017\)](#) defines the interface and standard behaviors for processes and threads.

Windows has a similar list of properties but sets its own standard. The Windows file permissions, directory structures, and process registry are very different from POSIX. Windows processes, represented by [Win32_Process](#), can be queried in WMI, the Windows Management Interface runtime, or by using the Task Manager.

Once a process is started on an Operating System, it is given:

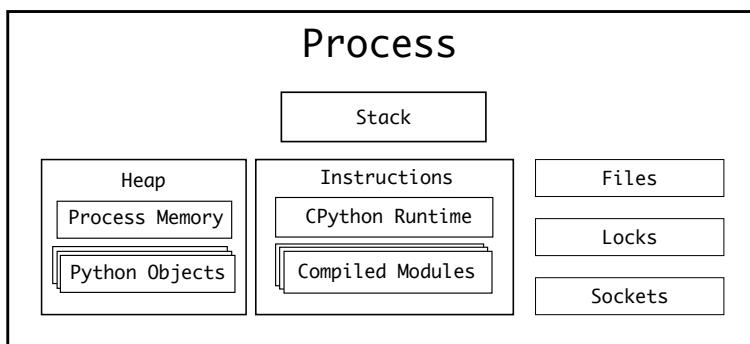
- A **Stack** of memory for calling subroutines
- A **Heap** (see Dynamic Memory Allocation in C)

- Access to **Files**, **Locks**, and **Sockets** on the Operating System

The CPU on your computer also keeps additional data when the process is executing, such as :

- **Registers** holding the current instruction being executed or any other data needed by the process for that instruction
- An **Instruction Pointer**, or **Program Counter** indicating which instruction in the program sequence is being executed

The CPython process comprises of the compiled CPython interpreter, and the compiled modules. These modules are loaded at runtime and converted into instructions by the CPython Evaluation Loop:



The program register and program counter point to a **single** instruction in the process. This means that only one instruction can be executing at any one time.

For CPython, this means that only one Python bytecode instruction can be executing at any one time.

There are two main approaches to allowing parallel execution of instructions in a process:

1. Fork another process

2. Spawn a thread

Now that you have reviewed what makes up a process. Next, you can explore forking and spawning child processes.

Multi-Process Parallelism

POSIX systems provide an API for any process to **fork** a child process.

Forking processes is a low-level API call to the Operating System that can be made by any running process.

When this call is made, the OS will clone all the attributes of the currently running process and create a new process.

This clone operation includes the heap, register, and counter position of the parent process. The child process can read any variables from the parent process at the time of forking.

Forking a Process in POSIX

As an example, take the Fahrenheit to Celcius example application used at the beginning of Dynamic Memory Allocation in C. Adapt it to spawn a child process for each Fahrenheit value instead of calculating them in sequence.

This is accomplished by using the `fork()` function. Each child process will continue operating from that point:

```
cpython-book-samples▶33▶thread_celcius.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static const double five_ninths = 5.0/9.0;

double celsius(double fahrenheit){
```

```

    return (fahrenheit - 32) * five_ninths;
}

int main(int argc, char** argv) {
    if (argc != 2)
        return -1;
    int number = atoi(argv[1]);
    for (int i = 1 ; i <= number ; i++ ) {
        double f_value = 100 + (i*10);
        pid_t child = fork();
        if (child == 0) { // Is child process
            double c_value = celsius(f_value);
            printf("%f F is %f C (pid %d)\n", f_value, c_value, getpid());
            exit(0);
        }
    }
    printf("Spawned %d processes from %dn", number, getpid());
    return 0;
}

```

Running this on the command-line would give an output similar to:

```

$ ./thread_celcius 4
110.000000 F is 43.333333 C (pid 57179)
120.000000 F is 48.888889 C (pid 57180)
Spawned 4 processes from 57178
130.000000 F is 54.444444 C (pid 57181)
140.000000 F is 60.000000 C (pid 57182)

```

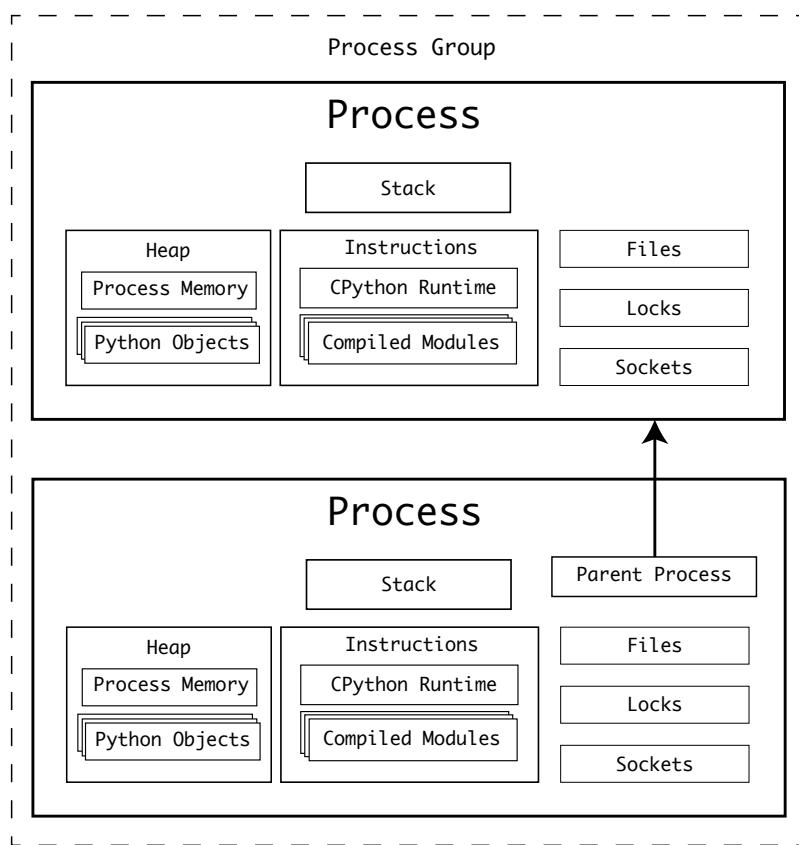
The parent process (57178), spawned 4 processes. For each child process, it continued at the line `child = fork()`, where the resulting value of `child` is 0. It then completes the calculation, prints the value, and exits the process.

Finally, the parent process outputs how many processes it spawned, and it's own PID.

The time taken for the 3rd and 4th child processes to complete was longer than it took for the parent process to complete. This is why

the parent process prints the final output before the 3rd and 4th print their own.

A parent process can exit, with its own exit code before a child process. Child Processes will be added to a Process Group by the Operating System, making it easier to control all related processes:



The biggest downside with this approach to parallelism is that the child process is a complete copy of the parent process.

In the case of CPython, this means you would have 2 CPython interpreters running, and both would have to load the modules and all the

libraries. It creates significant overhead. Using multiple processes makes sense when the overhead of forking a process is outweighed by the size of the task being completed.

Another major downside of forked processes is that they have a separate, isolated, heap from the parent process. This means that the child process cannot write to the memory space of the parent process. When creating the child process, the parent's heap becomes available to the child process. To send information back to the parent, some form of Inter-Process-Communication (IPC) must be used.

Note

The `os` module offers a wrapper around the `fork()` function.

Multi-Processing in Windows

So far, you've been reading the POSIX model. Windows doesn't provide an equivalent to `fork()`, and Python *should* (as best as possible) have the same API across Linux, macOS, and Windows.

To overcome this, the `CreateProcessW()` API is used to spawn another `python.exe` process with a `-c` command-line argument.

This step is known as "spawning," a process and is also available on POSIX. You'll see references to it throughout this chapter.

The `multiprocessing` Package

CPython provides an API on top of the Operating System process forking API. This API makes it simple to create multi-process parallelism in Python.

This API is available from the `multiprocessing` package. This package provides expansive capabilities for pooling processes, queues, forking, creating shared memory heaps, connecting processes together, and more.

Related Source Files

Source files related to multiprocessing are:

File	Purpose
Lib\multiprocessing	Python Source for the <code>multiprocessing</code> package
Modules__posixsubprocess.c	C extension module wrapping the POSIX <code>fork()</code> syscall
Modules__winapi.c	C extension module wrapping the Windows Kernel APIs
Modules__multiprocessing	C extension module used by the <code>multiprocessing</code> package
PC\msvcrtmodule.c	A Python interface to the Microsoft Visual C Runtime Library

Spawning and Forking Processes

The multiprocessing package offers three methods to start a new parallel process.

1. Forking an Interpreter (on POSIX only)
2. Spawning a new Interpreter process (on POSIX and Windows)
3. Running a Fork Server, where a new process is created which then forks any number of processes (on POSIX only)

Note

For Windows and macOS, the default start method is Spawning. For Linux, the default is Forking. You can override the default method using the `multiprocessing.set_start_method()` function.

The Python API for starting a new process takes a callable, `target`, and a tuple of arguments, `args`.

Take this simple example of spawning a new process to convert Fahrenheit to Celcius:

cpython-book-samples▶33▶spawn_process_celcius.py

```
import multiprocessing as mp
import os

def to_celcius(f):
    c = (f - 32) * (5/9)
    pid = os.getpid()
    print(f"{f}F is {c}C (pid {pid})")

if __name__ == '__main__':
    mp.set_start_method('spawn')
    p = mp.Process(target=to_celcius, args=(110,))
    p.start()
```

While you can start a single process, the `multiprocessing` API assumes you want to start multiple. There are convenience methods for spawning multiple processes and feeding them sets of data. One of those methods is the `Pool` class.

The previous example can be expanded to calculate a range of values in separate Python interpreters:

cpython-book-samples▶33▶pool_process_celcius.py

```
import multiprocessing as mp
import os

def to_celcius(f):
    c = (f - 32) * (5/9)
    pid = os.getpid()
    print(f"{f}F is {c}C (pid {pid})")

if __name__ == '__main__':
    mp.set_start_method('spawn')
    with mp.Pool(4) as pool:
        pool.map(to_celcius, range(110, 150, 10))
```

Note that the output shows the same PID. Because the CPython inter-

preter process has a signification overhead, the Pool will consider each process in the pool a “worker.” If a worker has completed, it will be reused. If you replace the line:

```
with mp.Pool(4) as pool:
```

with:

```
with mp.Pool(4, maxtasksperchild=1) as pool:
```

The previous multiprocessing example will print something similar to:

```
$ python pool_process_celcius.py
110F is 43.3333333333336C (pid 5654)
120F is 48.8888888888889C (pid 5653)
130F is 54.44444444444445C (pid 5652)
140F is 60.0C (pid 5655)
```

The output shows the process IDs of the newly spawned processes and the calculated values.

Creation of Child Processes

Both of these scripts will create a new Python interpreter process and pass data to it using pickle.

See Also

The `pickle` module is a serialization package used for serializing Python objects. Davide Mastromatteo has written a great write up of the [pickle module at realpython.com](https://realpython.com/pickle-in-python/).

For POSIX systems, the creation of the subprocess by the `multiprocessing` module is equivalent to this command:

```
$ python -c 'from multiprocessing.spawn import spawn_main;
spawn_main(tracker_fd=<i>, pipe_handle=<j>)' --multiprocessing-fork
```

Where `<i>` is the filehandle descriptor, and `<j>` is the pipe handle de-

scriptor.

For Windows systems, the parent PID is used instead of a tracker file descriptor:

```
> python.exe -c 'from multiprocessing.spawn import spawn_main;
    spawn_main(parent_pid=<k>, pipe_handle=<j>)' --multiprocessing-fork
```

Where <k> is the parent PID and <j> is the pipe handle descriptor.

Piping Data to the Child Process

When the new child process has been instantiated on the OS, it will wait for initialization data from the parent process.

The parent process writes 2 objects to a pipe file stream. The pipe file stream is a special IO stream used to send data between processes on the command line.

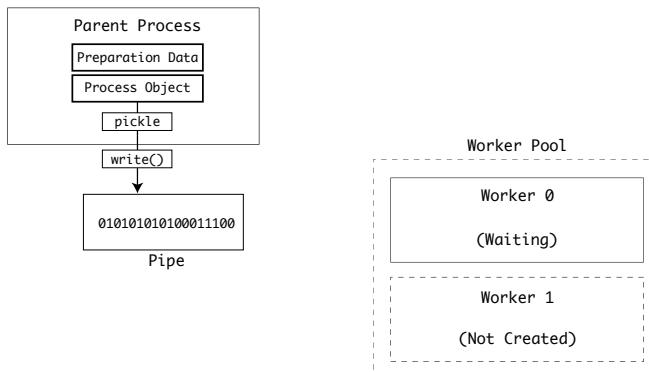
The first object written by the parent process is the **preparation data** object. This object is a dictionary containing some information about the parent, such as the executing directory, the start method, any special command-line arguments, and the sys.path. You can see an example of what is generated by running `multiprocessing.spawn.get_preparation_data(name)`:

```
>>> import multiprocessing.spawn
>>> import pprint
>>> pprint pprint(multiprocessing.spawn.get_preparation_data("example"))
{'authkey': b'x90xaa_x22[x18rixbcag]x93xfef5xe5@[wJx99p#x00'
     b'xced4)1j.xc3c',
 'dir': '/Users/anthonyshaw',
 'log_to_stderr': False,
 'name': 'example',
 'orig_dir': '/Users/anthonyshaw',
 'start_method': 'spawn',
 'sys_argv': [''],
 'sys_path': [
```

```
'/Users/anthonyshaw',
]}
```

The second object written is the `BaseProcess` child class instance. Depending on how `multiprocessing` was called and which Operating System is being used, one of the child classes of `BaseProcess` will be the instance serialized.

Both the preparation data and process object are serialized using the `pickle` module and written to the parent process' pipe stream:



Note

The POSIX implementation of the child process spawning and serialization process is located in `Lib` \blacktriangleright `multiprocessing` \blacktriangleright `popen_spawn_posix.py`. The Windows implementation is located in `Lib` \blacktriangleright `multiprocessing` \blacktriangleright `popen_spawn_win32.py`.

Executing the Child Process

The entry point of the child process, `multiprocessing.spawn.spawn_main()` takes the argument `pipe_handle` and either `parent_pid` for Windows or `tracked_fd` for POSIX:

```

def spawn_main(pipe_handle, parent_pid=None, tracker_fd=None):
    """
    Run code specified by data received over pipe
    """

    assert is_forking(sys.argv), "Not forking"

```

For Windows, the function will call the [OpenProcess API](#) of the parent PID.

This process object is used to create a filehandle, `fd`, of the parent process pipe:

```

if sys.platform == 'win32':
    import msvcrt
    import _winapi

    if parent_pid is not None:
        source_process = _winapi.OpenProcess(
            _winapi.SYNCHRONIZE | _winapi.PROCESS_DUP_HANDLE,
            False, parent_pid)
    else:
        source_process = None
        new_handle = reduction.duplicate(pipe_handle,
                                         source_process=source_process)
        fd = msvcrt.open_osfhandle(new_handle, os.O_RDONLY)
        parent_sentinel = source_process

```

For POSIX, the `pipe_handle` becomes the file descriptor, `fd`, and is duplicated to become the `parent_sentinel` value:

```

else:
    from . import resource_tracker
    resource_tracker._resource_tracker._fd = tracker_fd
    fd = pipe_handle
    parent_sentinel = os.dup(pipe_handle)

```

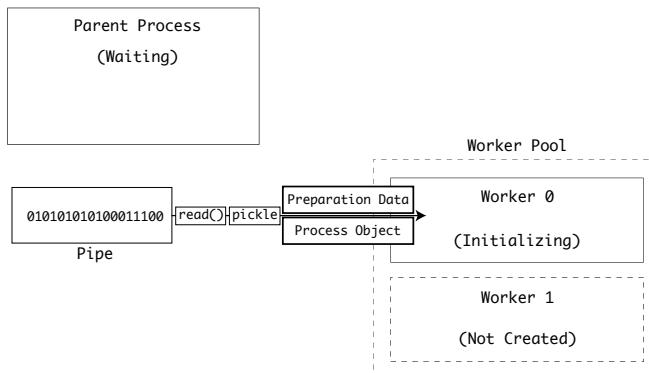
Next, the `_main()` function is called with the parent pipe file handle, `fd`, and the parent process sentinel, `parent_sentinel`. Whatever the return value of `_main()` is becomes the exit code for the process and the

interpreter is terminated:

```
exitcode = _main(fd, parent_sentinel)
sys.exit(exitcode)
```

The `_main()` function is called with the file descriptor of the parent processes pipe and the parent sentinel for checking if the parent process has exited whilst executing the child.

The main function deserialises the binary data on the `fa` byte stream. Remember, this is the `pipe` file handle. The deserialization happens using using same `pickle` library that the parent process used:



The first value is a `dict` containing the preparation data. The second value is an instance of `SpawnProcess` which is then used at the instance to call `_bootstrap()` upon:

```
def _main(fd, parent_sentinel):
    with os.fdopen(fd, 'rb', closefd=True) as from_parent:
        process.current_process()._inheriting = True
        try:
            preparation_data = reduction.pickle.load(from_parent)
            prepare(preparation_data)
            self = reduction.pickle.load(from_parent)
```

```
finally:  
    del process.current_process().__inheriting  
return self._bootstrap(parent_sentinel)
```

The `_bootstrap()` function handles the instantiation of a `BaseProcess` instance from the deserialized data, and then the target function is called with the arguments and keyword arguments. This final task is completed by `BaseProcess.run()`:

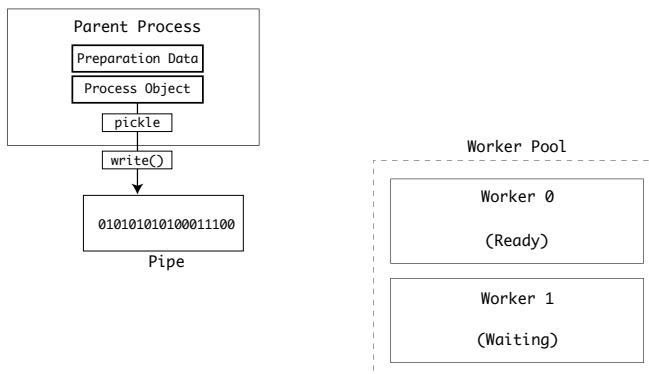
```
def run(self):  
    ...  
Method to be run in sub-process; can be overridden in sub-class  
    ...  
    if self._target:  
        self._target(*self._args, **self._kwargs)
```

The exit code of `self._bootstrap()` is set as the exit code, and the child process is terminated.

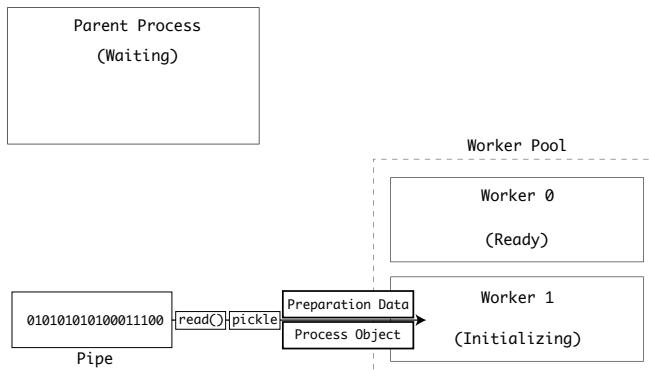
This process allows the parent process to serialize the module and the executable function. It also allows the child process to deserialize that instance, execute the function with arguments, and return.

It does not allow for the exchanging of data once the child process has started. This task is done using the extension of the `Queue` and `Pipe` objects.

If processes are being created in a pool, the first process will be ready and in a waiting state. The parent process repeats the process and sends the data to the next worker:



The next worker receives the data and initializes its state and runs the target function:



To share any data beyond initialization, queues and pipes must be used.

Exchanging Data with Queues and Pipes

In the previous section you saw how child processes are spawned, and then the pipe is used as a serialization stream to tell the child process what function to call with arguments.

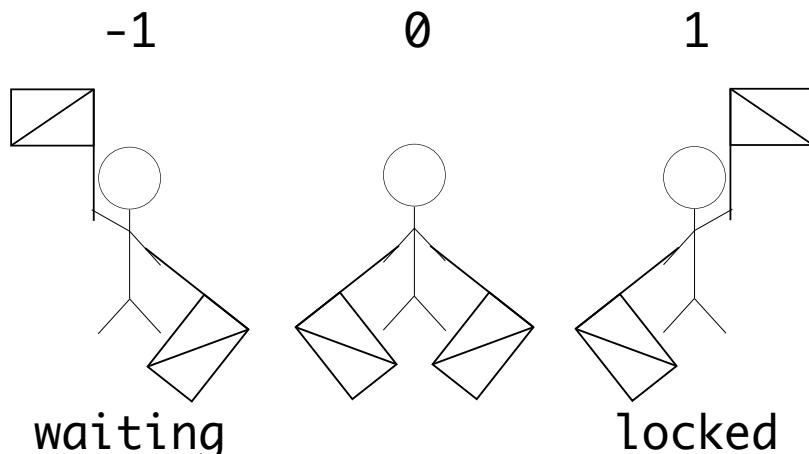
There are two types of communication between processes, depending on the nature of the task.

Semaphores

Many of the mechanisms in multiprocessing use **semaphores** as a way of signaling that resources are locked, being waited on, or not used. Operating Systems use binary semaphores as a simple variable type for locking resources, like files, sockets, and other resources.

If one process is writing to a file or a network socket, you don't want another process to suddenly start writing to the same file. The data would become corrupt instantly. Instead, Operating Systems put a "lock" on resources using a semaphore. Processes can also signal that they are waiting for that lock to be released so that when it is, they get a message to say it is ready and they can start using it.

Semaphores (in the real world) are a signaling method using flags, so the states for a resource of waiting, locked and not-used would look like:



The semaphore API is different between Operating Systems, so there is an abstraction class, `multiprocessing.synchronize.Semaphore`.

Semaphores are used by CPython for multiprocessing because they are both thread-safe and process-safe. The Operating System handles any potential deadlocks of reading or writing to the same semaphore.

The implementation of these semaphore API functions is located in a C extension module `Modules` \rightarrow `multiprocessing` \rightarrow `semaphore.c`. This extension module offers a single method for creating, locking, releasing semaphores, and other operations.

The call to the Operating System is through a series of Macros, which are compiled into different implementations depending on the Operating System platform. For Windows, the `<winbase.h>` API functions for semaphores are used:

```
#define SEM_CREATE(name, val, max) CreateSemaphore(NULL, val, max, NULL)
#define SEM_CLOSE(sem) (CloseHandle(sem)) ? 0 : -1
#define SEM_GETVALUE(sem, pval) _GetSemaphoreValue(sem, pval)
#define SEM_UNLINK(name) 0
```

For POSIX, the macros use the `<semaphore.h>` API is used:

```
#define SEM_CREATE(name, val, max) sem_open(name, O_CREAT | O_EXCL, 0600, val)
#define SEM_CLOSE(sem) sem_close(sem)
#define SEM_GETVALUE(sem, pval) sem_getvalue(sem, pval)
#define SEM_UNLINK(name) sem_unlink(name)
```

Queues

Queues are a great way of sending small data to and from multiple processes.

If you adapt the multiprocessing example before to use a `multiprocessing Manager()` instance, and create two queues:

1. `inputs` to hold the input Fahrenheit values
2. `outputs` to hold the resulting Celcius values

Change the pool size to 2 so that there are two workers:

```
cpython-book-samples▶33▶pool_queue_celcius.py
```

```
import multiprocessing as mp

def to_celcius(input: mp.Queue, output: mp.Queue):
    f = input.get()
    # time-consuming task ...
    c = (f - 32) * (5/9)
    output.put(c)

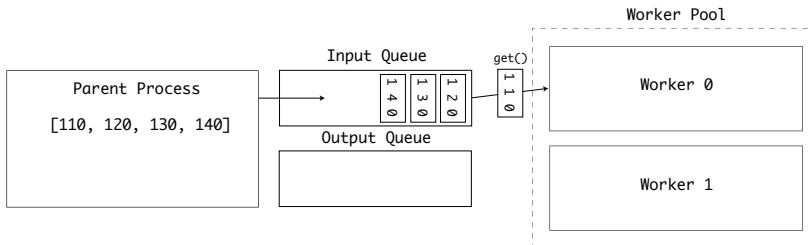
if __name__ == '__main__':
    mp.set_start_method('spawn')
    pool_manager = mp.Manager()
    with mp.Pool(2) as pool:
        inputs = pool_manager.Queue()
        outputs = pool_manager.Queue()
        input_values = list(range(110, 150, 10))
        for i in input_values:
            inputs.put(i)
        pool.apply(to_celcius, (inputs, outputs))

        for f in input_values:
            print(outputs.get(block=False))
```

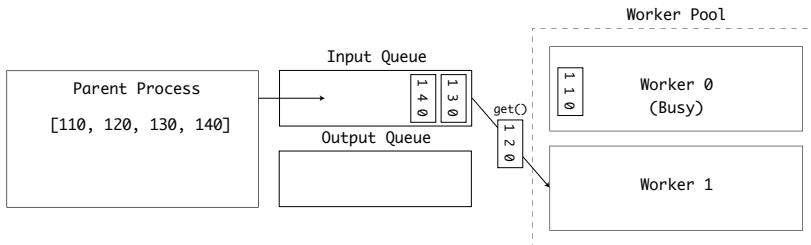
This would print the list of tuples returned to the `results` queue:

```
$ python pool_queue_celcius.py
43.33333333333336
48.8888888888889
54.44444444444445
60.0
```

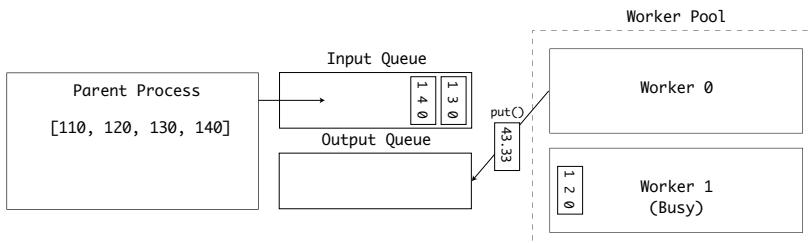
The parent process first puts the input values onto the input queue. The first worker then takes an item from the queue. Each time an item is taken from the queue using `.get()`, a semaphore lock is used on the queue object:



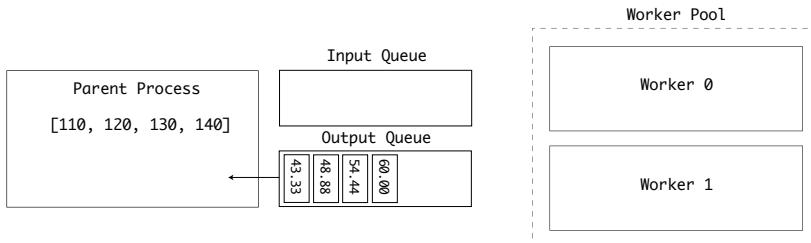
While this worker is busy, the second worker then takes another value from the queue:



The first worker has completed its calculation and puts the resulting value onto the result queue:



Two queues are in use to separate the input and output values. Eventually, all input values have been processed, and the output queue is full. The values are then printed by the parent process:

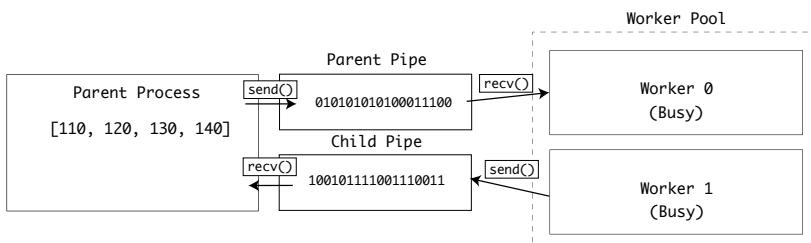


This example shows how a pool of workers could receive a queue of small, discreet values and process them in parallel to send the resulting data back to the host process. In practice, converting Celcius to Fahrenheit is a small, trivial calculation unsuited for parallel execution. If the worker process were doing another CPU-intensive calculation, this would provide significant performance improvement on a multi-CPU or multi-core computer.

For streaming data instead of discreet queues, pipes can be used instead.

Pipes

Within the `multiprocessing` package, there is a type `Pipe`. Instantiating a Pipe returns two connections, a parent and a child. Both can send and receive data:



In the queue example, a lock is implicitly placed on the queue when data is sent and received. Pipes do not have that behavior, so you have

to be careful that two processes do not try and write to the same pipe at the same time.

If you adapt the last example to work with a pipe, it will require changing the `pool.apply()` to `pool.apply_async()`. This changes the execution of the next process to a non-blocking operation:

```
cpython-book-samples▶ 33▶ pool_pipe_celcius.py
```

```
import multiprocessing as mp

def to_celcius(child_pipe: mp.Pipe, parent_pipe: mp.Pipe):
    f = parent_pipe.recv()
    # time-consuming task ...
    c = (f - 32) * (5/9)
    child_pipe.send(c)

if __name__ == '__main__':
    mp.set_start_method('spawn')
    pool_manager = mp.Manager()
    with mp.Pool(2) as pool:
        parent_pipe, child_pipe = mp.Pipe()
        results = []
        for i in range(110, 150, 10):
            parent_pipe.send(i)
            pool.apply_async(to_celcius, args=(child_pipe, parent_pipe))
            print(child_pipe.recv())
        parent_pipe.close()
        child_pipe.close()
```

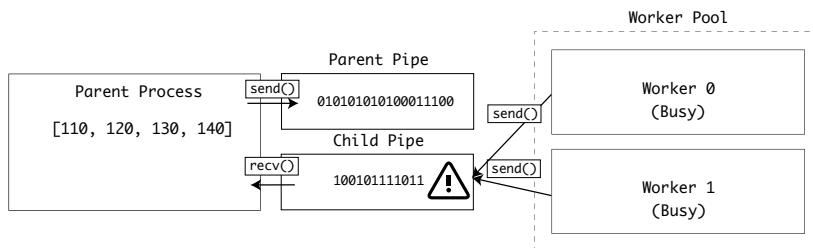
In this example, there is a risk of two or more processes trying to read from the parent pipe at the same time on the line:

```
f = parent_pipe.recv()
```

There is also a risk of two or more processes trying to write to the child pipe at the same time.

```
child_pipe.send(c)
```

If this situation occurs, data would be corrupted in either the receive or send operations:



To avoid this, you can implement a semaphore lock on the Operating System. Then all child processes will check with the Lock before reading or writing to the same pipe.

There are two locks required, one on the receiving end of the parent pipe, and another on the sending end of the child pipe:

```
cpython-book-samples▶33▶pool_pipe_locks_celcius.py
```

```
import multiprocessing as mp

def to_celcius(child_pipe: mp.Pipe, parent_pipe: mp.Pipe,
               child_write_lock: mp.Lock, parent_read_lock: mp.Lock):
    parent_read_lock.acquire()
    try:
        f = parent_pipe.recv()
    finally:
        parent_read_lock.release()
        # time-consuming task ...
        c = (f - 32) * (5/9)

    child_write_lock.acquire()
    try:
        child_pipe.send(c)
```

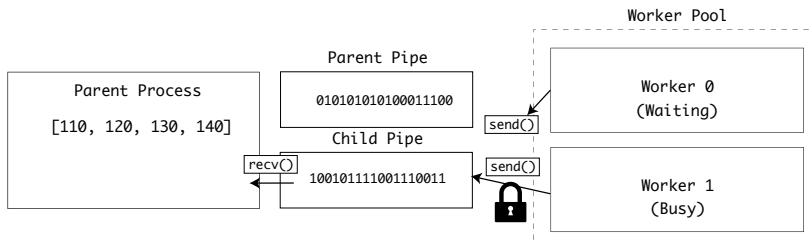
```

finally:
    child_write_lock.release()

if __name__ == '__main__':
    mp.set_start_method('spawn')
    pool_manager = mp.Manager()
    with mp.Pool(2) as pool:
        parent_pipe, child_pipe = mp.Pipe()
        parent_read_lock = mp.Lock()
        child_write_lock = mp.Lock()
        results = []
        for i in range(110, 150, 10):
            parent_pipe.send(i)
            pool.apply_async(to_celcius, args=(child_pipe, parent_pipe,
                                              child_write_lock,
                                              parent_read_lock))
            print(child_pipe.recv())
        parent_pipe.close()
        child_pipe.close()

```

Now the worker processes will wait to acquire a lock before receiving data, and wait again to acquire another lock to send data:



This example would suit situations where the data going over the pipe is large because the chance of a collision is higher.

Shared State Between Processes

So far, you have seen how data can be shared between the child and the parent process.

There may be scenarios where you want to share data between child processes. In this situation, the `multiprocessing` package provides two solutions:

1. A performant [Shared Memory API](#) using shared memory maps and shared C types
2. A flexible [Server Process API](#) supporting complex types via the `Manager` class

Example Application

As a demonstration application, throughout this chapter, you will be refactoring a TCP port scanner for different concurrency and parallelism techniques.

Over a network, a host can be contacted on ports, which are a number from 1-65535. Common services have standard ports. For example, HTTP operates on port 80 and HTTPS on 443. TCP port scanners are used as a common network testing tool to check that packets can be sent over a network.

This code example uses the `Queue` interface, a thread-safe queue implementation similar to the one you use in the `multiprocessing` examples. The code also uses the `socket` package to try connecting to a remote port with a short timeout of 1 second.

The `check_port()` function will see if the `host` responds on the given `port`, and if it does respond, it adds the port number to the `results` queue.

When the script is executed, the `check_port()` function is called in sequence for port numbers 80-100.

After this has completed, the results queue is emptied out, and the results are printed on the command line.

So you can compare the difference, it will print the execution time at the end:

```
cpython-book-samples▶33▶portscanner.py
```

```
from queue import Queue
import socket
import time
timeout = 1.0

def check_port(host: str, port: int, results: Queue):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.settimeout(timeout)
    result = sock.connect_ex((host, port))
    if result == 0:
        results.put(port)
    sock.close()

if __name__ == '__main__':
    start = time.time()
    host = "localhost" # replace with a host you own
    results = Queue()
    for port in range(80, 100):
        check_port(host, port, results)
    while not results.empty():
        print("Port {} is open".format(results.get()))
    print("Completed scan in {} seconds".format(time.time() - start))
```

The execution will print out the open ports and the time taken:

```
$ python portscanner.py
Port 80 is open
Completed scan in 19.623435020446777 seconds
```

This example can be refactored to use multiprocessing. The Queue interface is swapped for `multiprocessing.Queue` and the ports are scanned together using a pool executor:

```
cpython-book-samples▶33▶portscanner_mp_queue.py
```

```

import multiprocessing as mp
import time
import socket

timeout = 1

def check_port(host: str, port: int, results: mp.Queue):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.settimeout(timeout)
    result = sock.connect_ex((host, port))
    if result == 0:
        results.put(port)
    sock.close()

if __name__ == '__main__':
    start = time.time()
    processes = []
    scan_range = range(80, 100)
    host = "localhost" # replace with a host you own
    mp.set_start_method('spawn')
    pool_manager = mp.Manager()
    with mp.Pool(len(scan_range)) as pool:
        outputs = pool_manager.Queue()
        for port in scan_range:
            processes.append(pool.apply_async(check_port,
                                              (host, port, outputs)))
    for process in processes:
        process.get()
    while not outputs.empty():
        print("Port {} is open".format(outputs.get()))
    print("Completed scan in {} seconds".format(time.time() - start))

```

As you might expect, this application is much faster because it is testing each port in parallel:

```

$ python portscanner_mp_queue.py
Port 80 is open
Completed scan in 1.556523084640503 seconds

```

Conclusion

Multiprocessing offers a scalable, parallel execution API for Python. Data can be shared between processes, and CPU-intensive work can be broken into parallel tasks to take advantage of multiple core or CPU computers.

Multiprocessing is not a suitable solution when the task to be completed is not CPU intensive, but instead IO-bound. For example, if you spawned 4 worker processes to read and write to the same files, one would do all the work, and the other 3 would wait for the lock to be released.

Multiprocessing is also not suitable for short-lived tasks, because of the time and processing overhead of starting a new Python interpreter.

In both of those scenarios, you may find one of the next approaches is more suited.

Multithreading

C_Python provides a high-level and a low-level API for creating, spawning, and controlling threads from Python.

To understand Python threads, you should first understand how Operating System threads work. There are two implementations of threading in C_Python.

1. `pthreads` - POSIX threads for Linux and macOS
2. `nt threads` - NT threads for Windows

In the section on The Structure of a Process, you saw how a process has:

- A **Stack** of subroutines
- A **Heap** of memory

- Access to **Files**, **Locks**, and **Sockets** on the Operating System

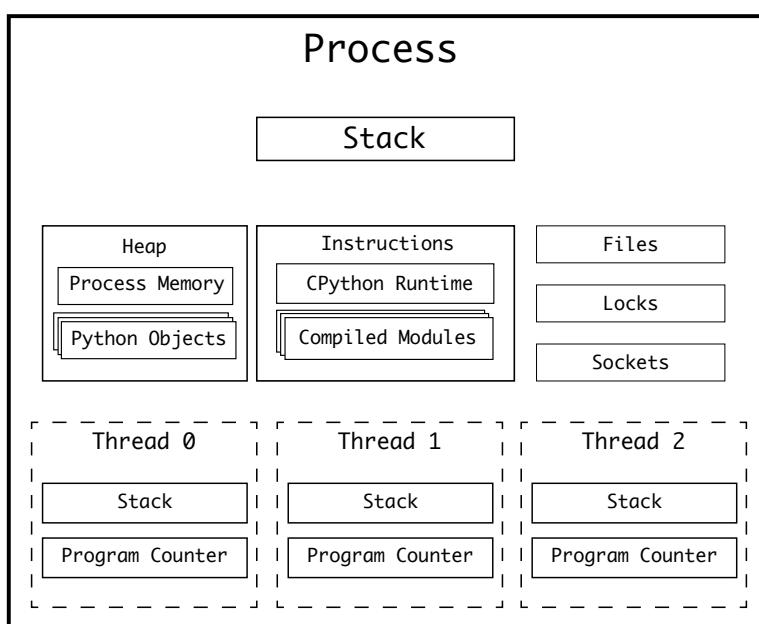
The biggest limitation to scaling a single process is that the Operating System will have a single **Program Counter** for that executable.

To get around this, modern Operating Systems allow processes to signal the Operating System to branch their execution into multiple threads.

Each thread will have its own Program Counter, but use the same resources as the host process. Each thread also has its own call stack, so it can be executing a different function.

Because multiple threads can read and write to the same memory space, collisions could occur. The solution to this is **thread safety** and involves making sure that memory space is locked by a single thread before it is accessed.

A single process with 3 threads would have a structure:



See Also

For a great introductory tutorial on the Python threading API, check out [Jim Anderson's "Intro to Python Threading."](#)

The GIL

If you're familiar with NT threads or POSIX threads from C, or you've used another high-level language, you may expect multithreading to be parallel.

In CPython, the threads are based on the C APIs, but the threads are Python threads. This means that every Python thread needs to execute Python bytecode through the evaluation loop.

The Python evaluation loop is not thread-safe. There are many parts of the interpreter state, such as the Garbage Collector, which are shared, and global.

To get around this, the CPython developers implemented a mega-lock, called the **Global Interpreter Lock (GIL)**. Before any opcode is executed in the frame-evaluation loop, the GIL is acquired by the thread, then once the opcode has been executed, it is released.

Aside from providing a global thread-safety to every operation in Python, this approach has a major drawback. Any operations which take a long time to execute will leave other threads waiting for the GIL to be released before they can execute.

This means that only 1 thread can be executing a Python bytecode operation at any one time.

To acquire the GIL, a call is made to `take_gil()` and then again to `drop_gil()` to release it. The GIL acquisition is made within the core frame evaluation loop, `_PyEval_EvalFrameDefault()`.

To stop a single frame execution from permanently holding the GIL, the evaluation loop state stores a flag, `gil_drop_request`. After every

bytecode operation has completed in a frame, this flag is checked, and the GIL is temporarily released and then reacquired:

```

if (_Py_atomic_load_relaxed(&ceval->gil_drop_request)) {
    /* Give another thread a chance */
    if (_PyThreadState_Swap(&runtime->gilstate, NULL) != tstate) {
        Py_FatalError("ceval: tstate mix-up");
    }
    drop_gil(ceval, tstate);

    /* Other threads may run now */

    take_gil(ceval, tstate);

    /* Check if we should make a quick exit. */
    exit_thread_if_finalizing(tstate);

    if (_PyThreadState_Swap(&runtime->gilstate, tstate) != NULL) {
        Py_FatalError("ceval: orphan tstate");
    }
}
...

```

Despite the limitations that the GIL enforces on parallel execution, it means that multithreading in Python is very safe and ideal for running IO-bound tasks concurrently.

Related Source Files

Source files related to threading are:

File	Purpose
Include▶pythread.h	PyThread API and definition
Lib▶threading.py	High Level threading API and Standard Library module
Modules▶_threadmodule.c	Low Level thread API and Standard Library module
Python▶thread.c	C extension for the <code>thread</code> module
Python▶thread_nt.h	Windows Threading API
Python▶thread_pthread.h	POSIX Threading API

File	Purpose
Python▶ceval_gil.h	GIL lock implementation

Starting Threads in Python

To demonstrate the performance gains of having multithreaded code (in spite of the GIL), you can implement a simple network port scanner in Python.

Now clone the previous script but change the logic to spawn a thread for each port using `threading.Thread()`. This is similar to the `multiprocessing` API, where it takes a callable, `target`, and a tuple, `args`. Start the threads inside the loop, but don't wait for them to complete. Instead, append the thread instance to a list, `threads`:

```
for port in range(800, 100):
    t = Thread(target=check_port, args=(host, port, results))
    t.start()
    threads.append(t)
```

Once all threads have been created, iterate through the thread list and call `.join()` to wait for them to complete:

```
for t in threads:
    t.join()
```

Next, exhaust all the items in the `results` queue and print them to the screen:

```
while not results.empty():
    print("Port {0} is open".format(results.get()))
```

The whole script is:

cpython-book-samples▶33▶portscanner_threads.py

```
from threading import Thread
from queue import Queue
```

```

import socket
import time

timeout = 1.0

def check_port(host: str, port: int, results: Queue):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.settimeout(timeout)
    result = sock.connect_ex((host, port))
    if result == 0:
        results.put(port)
    sock.close()

def main():
    start = time.time()
    host = "localhost" # replace with a host you own
    threads = []
    results = Queue()
    for port in range(80, 100):
        t = Thread(target=check_port, args=(host, port, results))
        t.start()
        threads.append(t)
    for t in threads:
        t.join()
    while not results.empty():
        print("Port {} is open".format(results.get()))
    print("Completed scan in {} seconds".format(time.time() - start))

if __name__ == '__main__':
    main()

```

When you call this threaded script at the command-line, it will execute 10+ times faster than the single-threaded example:

```

$ python portscanner_threads.py
Port 80 is open
Completed scan in 1.0101029872894287 seconds

```

This also runs 50-60% faster than the multiprocessing example. Re-

member that multiprocessing has an overhead for starting the new processes, threading does have an overhead, but it is much smaller.

You may be wondering- if the GIL means that only a single operation can execute at once, why is this faster?

The statement that takes 1-1000ms is:

```
result = sock.connect_ex((host, port))
```

In the C extension module, `Modules` ▶ `socketmodule.c`, the function that implements the connection is:

`Modules` ▶ `socketmodule.c` line 3246

```
static int
internal_connect(PySocketSockObject *s, struct sockaddr *addr, int addrlen,
                  int raise)
{
    int res, err, wait_connect;

    Py_BEGIN_ALLOW_THREADS
    res = connect(s->sock_fd, addr, addrlen);
    Py_END_ALLOW_THREADS
```

Surrounding the system `connect()` call are the `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros.

These macros are defined in `Include` ▶ `ceval.h` as:

```
#define Py_BEGIN_ALLOW_THREADS {
    PyThreadState *_save;
    _save = PyEval_SaveThread();
#define Py_BLOCK_THREADS      PyEval_RestoreThread(_save);
#define Py_UNBLOCK_THREADS    _save = PyEval_SaveThread();
#define Py_END_ALLOW_THREADS  PyEval_RestoreThread(_save);
}
```

So, when `Py_BEGIN_ALLOW_THREADS` is called, it calls `PyEval_SaveThread()`. This function changes the thread state to `NULL` and **drops** the GIL:

Python▶ ceval.c line 480

```
PyThreadState *
PyEval_SaveThread(void)
{
    PyThreadState *tstate = PyThreadState_Swap(NULL);
    if (tstate == NULL)
        Py_FatalError("PyEval_SaveThread: NULL tstate");
    assert(gil_created());
    drop_gil(tstate);
    return tstate;
}
```

Because the GIL is dropped, it means any other executing thread can continue. This thread will sit and wait for the system call without blocking the evaluation loop.

Once the `connect()` function has succeeded or timed out, the `Py_END_ALLOW_THREADS` runs the `PyEval_RestoreThread()` function with the original thread state.

The thread state is recovered and the GIL is retaken. The call to `take_gil()` is a blocking call, waiting on a semaphore:

Python▶ ceval.c line 503

```
void
PyEval_RestoreThread(PyThreadState *tstate)
{
    if (tstate == NULL)
        Py_FatalError("PyEval_RestoreThread: NULL tstate");
    assert(gil_created());

    int err = errno;
    take_gil(tstate);
    /* _Py_Finalizing is protected by the GIL */
    if (_Py_IsFinalizing() && !_Py_CURRENTLY_FINALIZING(tstate)) {
        drop_gil(tstate);
        PyThread_exit_thread();
```

```
    Py_UNREACHABLE();  
}  
errno = err;  
  
PyThreadState_Swap(tstate);  
}
```

This is not the only system call wrapped by the non-GIL-blocking pair `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS`. There are over 300 uses of it in the Standard Library. Including:

- Making HTTP requests
- Interacting with local hardware
- Encryption
- Reading and writing files

Thread State

C^oPython provides its own implementation of thread management. Because threads need to execute Python bytecode in the evaluation loop, running a thread in C^oPython isn't as simple as spawning an OS thread. Python threads are called `PyThread`, and you covered them briefly on the C^oPython Evaluation Loop chapter.

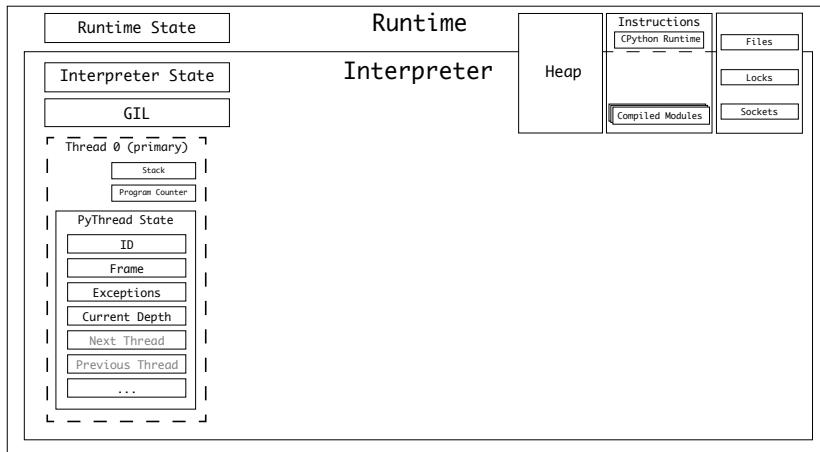
Python threads execute code objects and are spawned by the interpreter.

To recap:

- C^oPython has a single runtime, which has its own **runtime state**
- C^oPython can have one or many interpreters
- An interpreter has a state, called the **interpreter state**
- An interpreter will take a **code object** and convert it into a series of **frame objects**
- An interpreter has at least one **thread**, each thread has a **thread state**

- Frame Objects are executed in a stack, called the **frame stack**
- CPython references variables in a **value stack**
- The **interpreter state** includes a linked-list of its threads

A single-threaded, single-interpreter runtime would have the states:



The thread state type, `PyThreadState` has over 30 properties, including:

- A unique identifier
- A linked-list to the other thread states
- The interpreter state it was spawned by
- The currently executing frame
- The current recursion depth
- Optional tracing functions
- The exception currently being handled
- Any async exception currently being handled
- A stack of exceptions raised
- A GIL counter

- Async generator counters

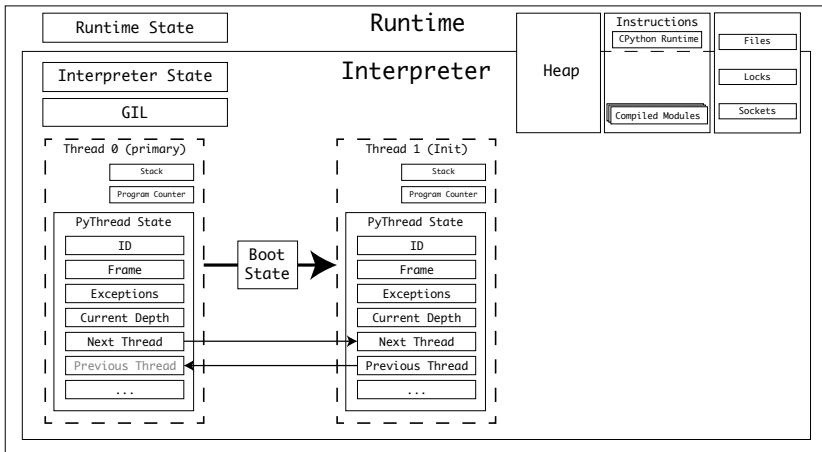
Similar to the multiprocessing **preparation data**, threads have a boot state. However, threads share the same memory space, so there is no need to serialize data and send it over a file stream.

Threads are instantiated with the `threading.Thread` type. This is a high-level module that abstracts the `PyThread` type. `PyThread` instances are managed by the C extension module `_thread`.

The `_thread` module has the entry point for executing a new thread, `thread_PyThread_start_new_thread()`. `start_new_thread()` is a method on an instance of the type `Thread`.

New threads are instantiated in this sequence:

1. The `bootstate` is created, linking to the target, with arguments `args` and `kwargs`
2. The `bootstate` is linked to the interpreter state
3. A new `PyThreadState` is created, linking to the current interpreter
4. The GIL is enabled, if not already with a call to `PyEval_InitThreads()`
5. The new thread is started on the Operating System-specific implementation of `PyThread_start_new_thread`



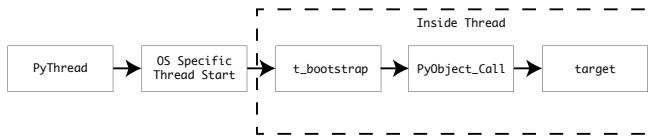
Thread `bootstate` has the properties:

Field	Type	Purpose
<code>interp</code>	<code>PyInterpreterState*</code>	Link to the interpreter managing this thread
<code>func</code>	<code>PyObject * (callable)</code>	Link to the callable to execute upon running the thread
<code>args</code>	<code>PyObject * (tuple)</code>	Arguments to call <code>func</code> with
<code>keyw</code>	<code>PyObject * (dict)</code>	Keyword arguments to call <code>func</code> with
<code>tstate</code>	<code>PyThreadState *</code>	Thread state for the new thread

With the thread `bootstate`, there are two implementations `PyThread` - POSIX threads for Linux and macOS, and NT threads for Windows.

Both of these implementations create the Operating System thread, set its attribute and then execute the callback `t_bootstrap()` from within the new thread. This function is called with the single argument `boot_raw`, assigned to the `bootstate` constructed in `thread_PyThread_start_new_thread()`.

The `t_bootstrap()` function is the interface between a low-level thread and the Python runtime. The bootstrap will initialize the thread, then execute the `target` callable using `PyObject_Call()`. Once the callable target has been executed, the thread will exit:



POSIX Threads

POSIX threads, named `pthreads`, have an implementation in `Python/thread_pthread.h`. This implementation abstracts the `<pthread.h>` C API with some additional safeguards and optimizations.

Threads can have a configured stack size. Python has its own stack frame construct, as you explored in the chapter on the Evaluation Loop. If there is an issue causing a recursive loop, and the frame execution hits the depth limit, Python will raise a `RecursionError` which can be handled from a `try..except` block in Python code. Because `pthreads` have their own stack size, the max depth of Python and the stack size of the `pthread` might conflict.

If the thread stack size is smaller than the max frame depth in Python, the entire Python process will crash before a `RecursionError` is raised. Also, the max depth in Python can be configured at runtime using `sys.setrecursionlimit()`.

To avoid these crashes, the CPython `pthread` implementation sets the stack size to the `pythread_stacksize` value of the Interpreter State.

Most modern POSIX-compliant Operating Systems support system scheduling of `pthreads`. If `PTHREAD_SYSTEM_SCHED_SUPPORTED` is defined in `pyconfig.h`, the `pthread` is set to `PTHREAD_SCOPE_SYSTEM`, meaning that the priority of the thread on the Operating System scheduler is decided against the other threads on the system, not just the ones within the Python process.

Once the thread properties have been configured, the thread is created using the `pthread_create()` API. This runs the bootstrap function from inside the new thread.

Lastly, the thread handle, `pthread_t` is cast into an `unsigned long` and returned to become the thread ID.

Windows Threads

Windows threads implemented in `Python\thread_nt.h` follow a similar, but simpler pattern.

The stack size of the new thread is configured to the interpreter `pythread_stacksize` value (if set).

The thread is created using the `_beginthreadex()` Windows API using the bootstrap function as the callback.

Lastly, the thread ID is returned.

Conclusion

This was not an exhaustive tutorial on Python threads. Python's thread implementation is extensive and offers many mechanisms for sharing data between threads, locking objects, and resources.

Threads are a great, efficient way of improving the runtime of your Python applications when they are IO-bound. In this section, you have seen what the GIL is, why it exists and which parts of the standard library may be exempt from its constraints.

Asynchronous Programming

Python offers many ways of accomplishing concurrent programming without using threads or multiprocessing. These features have been added, expanded, and often replaced with better alternatives.

For the target version of this book, 3.9.0b1, the following asynchronous systems are deprecated:

- The `@coroutine` decorator

The following systems are still available:

- Creating futures from `async` keywords
- Coroutines using the `yield from` keywords

Generators

Python Generators are functions that return a `yield` statement and can be called continually to generate further values.

Generators are often used as a more memory efficient way of looping through values in a large block of data, like a file, a database, or over a network. Generator objects are returned in place of a **value** when `yield` is used instead of `return`. The generator object is created from the `yield` statement and returned to the caller.

This simple generator function will yield the letters a-z:

```
cpython-book-samples▶33▶letter_generator.py
```

```
def letters():
    i = 97  # letter 'a' in ASCII
    end = 97 + 26  # letter 'z' in ASCII
    while i < end:
        yield chr(i)
        i += 1
```

If you call `letters()`, it won't return a value, but instead it returns a generator object:

```
>>> from letter_generator import letters
>>> letters()
<generator object letters at 0x1004d39b0>
```

Built into the syntax of the `for` statement is the ability to iterate through a generator object until it stops yielding values:

```
>>> for letter in letters():
...     print(letter)
a
b
c
d
...
```

This implementation uses the iterator protocol. Objects that have a `__next__()` method can be looped over by `for` and `while` loops, or using the `next()` builtin.

All container types (like lists, sets, tuples) in Python implement the iterator protocol. Generators are unique because the implementation of the `__next__()` method recalls the generator function from its last state. Generators are not executing in the background, they are paused. When you request another value, they resume execution.

Within the generator object structure is the frame object as it was at the last `yield` statement.

Generator Structure

Generator objects are created by a template macro, `_PyGenObject_-HEAD(prefix)`.

This macro is used by the following types and prefixes:

1. `PyGenObject - gi_` (Generator objects)
2. `PyCoroObject - cr_` (Coroutine objects)
3. `PyAsyncGenObject - ag_` (Async generator objects)

You will cover coroutine and async generator objects later in this chapter.

The `PyGenObject` type has the base properties:

Field	Type	Purpose
Field	Type	Purpose
[x]_frame	PyFrameObject*	Current frame object for the generator
[x]_running	char	Set to 0 or 1 if the generator is currently running
[x]_code	PyObject * (PyCodeObject*)	Compiled function that yielded the generator
[x]_-weakreflist	PyObject * (list)	List of weak references to objects inside the generator function
[x]_name	PyObject * (str)	Name of the generator
[x]_qualname	PyObject * (str)	Qualified name of the generator
[x]_exc_-state	_PyErr_StackItem	Exception data if the generator call raises an exception

On top of the base properties, the `PyCoroObject` type has:

Field	Type	Purpose
cr_origin	PyObject * (tuple)	Tuple containing the originating frame and caller

On top of the base properties, the `PyAsyncGenObject` type has:

Field	Type	Purpose
ag_finalizer	PyObject *	Link to the finalizer method
ag_hooks_initied	int	Flag to mark that the hooks have been initialized
ag_closed	int	Flag to mark that the generator is closed
ag_running_async	int	Flag to mark that the generator is running

Related Source Files

Source files related to generators are:

File	Purpose
Include\genobject.h	Generator API and <code>PyGenObject</code> definition

File	Purpose
Objects\genobject.c	Generator Object implementation

Creating Generators

When a function containing a `yield` statement is compiled, the resulting code object has an additional flag, `CO_GENERATOR`.

In the chapter on the Execution Loop: Constructing Frames, you explored how a compiled code object is converted into a frame object when it is executed.

In the process, there is a special case for generators, coroutines, and async generators. The `_PyEval_EvalCode()` function checks the code object for the `CO_GENERATOR`, `CO_COROUTINE`, and `CO_ASYNC_GENERATOR` flags.

Instead of evaluation a code object inline, the frame is created and turned into a Generator, Coroutine or Async Generator Object. A coroutine is created using `PyCoro_New()`, an async generator is created with `PyAsyncGen_New()`, and a generator with `PyGen_NewWithQualName()`:

```
PyObject *
(PyEval_EvalCode(PyObject *_co, PyObject *globals, PyObject *locals, ...
...
/* Handle generator/coroutine/asynchronous generator */
if (co->co_flags & (CO_GENERATOR | CO_COROUTINE | CO_ASYNC_GENERATOR)) {
    PyObject *gen;
    PyObject *coro_wrapper = tstate->coroutine_wrapper;
    int is_coro = co->co_flags & CO_COROUTINE;
    ...
    /* Create a new generator that owns the ready to run frame
     * and return that as the value. */
    if (is_coro) {
        gen = PyCoro_New(f, name, qualname);
    } else if (co->co_flags & CO_ASYNC_GENERATOR) {
        gen = PyAsyncGen_New(f, name, qualname);
    } else {
        gen = PyGen_NewWithQualName(f, name, qualname);
    }
}
```

```
    }
    ...
    return gen;
}
...
```

The generator factory, `PyGen_NewWithQualifiedName()`, takes the frame and completes some steps to populate the generator object fields:

1. Sets the `gi_code` property to the compiled code object
2. Sets the generator to not running (`gi_running = 0`)
3. Sets the exception and weakref lists to NULL

You can also see that `gi_code` is the compiled code object for the generator function by importing the `dis` module and disassembling the bytecode inside:

```
>>> from letter_generator import letters
>>> gen = letters()
>>> import dis
>>> dis.disco(gen.gi_code)
 2           0 LOAD_CONST               1 (97)
 2           2 STORE_FAST                0 (i)
...
...
```

In the chapter on the Evaluation Loop, you explored the Frame Object Type. Frame objects contain locals and globals, the last executed instructions, and the code to be executed.

The builtin behavior and state of the frame object are how generators can *pause* and be *resumed* on demand.

Executing Generators

Whenever `__next__()` is called on a generator object, `gen_iternext()` is called with the generator instance, which immediately calls `gen_send_ex()` inside `Objects/genobject.c`.

`gen_send_ex()` is the function that converts a generator object into the next yielded result. You'll see many similarities with the way frames are constructed from a code object as these functions have similar tasks.

The `gen_send_ex()` function is shared with generators, coroutines, and async generators and has the following steps:

1. The current thread state is fetched
2. The frame object from the generator object is fetched
3. If the generator is running when `__next__()` was called, raise a `ValueError`
4. If the frame inside the generator is at the top of the stack:
 - In the case of a coroutine, if the coroutine is not already marked as closing, a `RuntimeError` is raised
 - If this is an async generator, raise a `StopAsyncIteration`
 - For a standard generator, a `StopIteration` is raised.
5. If the last instruction in the frame (`f->f_lasti`) is still `-1` because it has just been started, and this is a coroutine or async generator, then a non-`None` value can't be passed as an argument, so an exception is raised
6. Else, this is the first time it's being called, and arguments are allowed. The value of the argument is pushed to the frame's value stack
7. The `f_back` field of the frame is the caller to which return values are sent, so this is set to the current frame in the thread. This means that the return value is sent to the caller, not the creator of the generator
8. The generator is marked as running
9. The last exception in the generator's exception info is copied from the last exception in the thread state
10. The thread state exception info is set to the address of the generator's exception info. This means that if the caller enters a break-

point around the execution of a generator, the stack trace goes through the generator and the offending code is clear

11. The frame inside the generator is executed within the `Python->ceval.c` main execution loop, and the value returned
12. The thread state last exception is reset to the value before the frame was called
13. The generator is marked as not running
14. The following cases then match the return value and any exceptions thrown by the call to the generator. Remember that generators should raise a `StopIteration` when they are exhausted, either manually, or by not yielding a value. Coroutines and async generators should not:
 - If no result was returned from the frame, raise a `StopIteration` for generators and `StopAsyncIteration` for async generators
 - If a `StopIteration` was explicitly raised, but this is a coroutine or an async generator, raise a `RuntimeError` as this is not allowed
 - If a `StopAsyncIteration` was explicitly raised and this is an async generator, raise a `RuntimeError`, as this is not allowed
15. Lastly, the result is returned back to the caller of `__next__()`

Bringing this all together, you can see how the generator expression is a powerful syntax where a single keyword, `yield` triggers a whole flow to create a unique object, copy a compiled code object as a property, set a frame, and store a list of variables in the local scope.

Coroutines

Generators have a big limitation. They can only yield values to their immediate caller.

An additional syntax was added to Python to overcome this- the `yield from` statement. Using this syntax, you can refactor generators into utility functions and then `yield from` them.

For example, the letter generator can be refactored into a utility function where the starting letter is an argument. Using `yield from`, you can choose which generator object to return:

```
cpython-book-samples▶33▶letter_coroutines.py
```

```
def gen_letters(start, x):
    i = start
    end = start + x
    while i < end:
        yield chr(i)
        i += 1

def letters(upper):
    if upper:
        yield from gen_letters(65, 26) # A-Z
    else:
        yield from gen_letters(97, 26) # a-z

for letter in letters(False):
    # Lower case a-z
    print(letter)

for letter in letters(True):
    # Upper case A-Z
    print(letter)
```

Generators are also great for lazy sequences, where they can be called multiple times.

Building on the behaviors of generators, such as being able to pause and resume execution, the concept of a coroutine was iterated in Python over multiple APIs. Generators are a limited form of coroutine because you can send data to them using the `.send()` method.

It is possible to send messages bi-directionally between the caller and the target. Coroutines also store the caller in the `cr_origin` attribute.

Coroutines were initially available via a decorator, but this has since been deprecated in favor of “native” coroutines using the keywords `async` and `await`.

To mark that a function returns a coroutine, it must be preceded with the `async` keyword.

The `async` keyword makes it explicit (unlike generators) that this function returns a coroutine and not a value.

To create a coroutine, define a function with the keyword `async def`. In this example, add a timer using the `asyncio.sleep()` function and return a wake-up string:

```
>>> import asyncio
>>> async def sleepy_alarm(time):
...     await asyncio.sleep(time)
...     return "wake up!"
>>> alarm = sleepy_alarm(10)
>>> alarm
<coroutine object sleepy_alarm at 0x1041de340>
```

When you call the function, it returns a coroutine object. There are many ways to execute a coroutine. The easiest is using `asyncio.run(coro)`.

Run `asyncio.run()` with your coroutine object, then after 10 seconds it will sound the alarm:

```
>>> asyncio.run(alarm)
'wake up'
```

So far, there is a small benefit over a regular function. The benefit of coroutines is that you can run them concurrently. Because the coroutine object is a variable that you can pass to a function, these objects can be linked together and chained, or created in a sequence.

For example, if you wanted to have 10 alarms with different intervals and start them all at the same time, these coroutine objects can be

converted into tasks.

The task API is used to schedule and execute multiple coroutines concurrently.

Before tasks are scheduled, an event loop must be running. The job of the event loop is to schedule concurrent tasks and connect events such as completion, cancellation, and exceptions with callbacks.

When you called `asyncio.run()`, the `run` function (in `Lib` ➤ `asyncio` ➤ `runners.py`) did these tasks for you:

1. Started a new event loop
2. Wrapped the coroutine object in a task
3. Set a callback on the completion of the task
4. Looped over the task until it completed
5. Returned the result

Related Source Files

Source files related to coroutines are:

File	Purpose
<code>Lib</code> ➤ <code>asyncio</code>	Python standard library implementation for <code>asyncio</code>

Event Loops

Event loops are the glue that holds `async` code together. Written in pure Python, event loops are an object containing tasks.

When started, a loop can either run once or run forever. Any of the tasks in the loop can have callbacks. The loop will run the callbacks if a task completes or fails.

```
loop = asyncio.new_event_loop()
```

Inside a loop is a sequence of tasks, represented by the type `asyncio.Task`, tasks are scheduled onto a loop, then once the loop is running, it loops over all the tasks until they complete.

You can convert the single timer into a task loop:

```
cpython-book-samples▶33▶sleepy_alarm.py
```

```
import asyncio

async def sleepy_alarm(person, time):
    await asyncio.sleep(time)
    print(f"{person} -- wake up!")

async def wake_up_gang():
    tasks = [
        asyncio.create_task(sleepy_alarm("Bob", 3), name="wake up Bob"),
        asyncio.create_task(sleepy_alarm("Sanjeet", 4), name="wake up Sanjeet"),
        asyncio.create_task(sleepy_alarm("Doris", 2), name="wake up Doris"),
        asyncio.create_task(sleepy_alarm("Kim", 5), name="wake up Kim")
    ]
    await asyncio.gather(*tasks)

asyncio.run(wake_up_gang())
```

This will print:

```
Doris -- wake up!
Bob -- wake up!
Sanjeet -- wake up!
Kim -- wake up!
```

In the event loop, it will run over each of the coroutines to see if they are completed. Similar to how the `yield` keyword can return multiple values from the same frame, the `await` keyword can return multiple states. The event loop will execute the `sleepy_alarm()` coroutine objects again and again until the `await asyncio.sleep()` yields a completed re-

sult, and the `print()` function is able to execute.

For this to work, `asyncio.sleep()` must be used instead of the blocking (and not async-aware) `time.sleep()`.

Example

You can convert the multithreaded port scanner example to `asyncio` with these steps:

- Change the `check_port()` function to use a socket connection from `asyncio.open_connection()`, which creates a future instead of an immediate connection
- Use the socket connection future in a timer event, with `asyncio.wait_for()`
- Append the port to the results list if succeeded
- Add a new function, `scan()` to create the `check_port()` coroutines for each port and add them to a list, `tasks`
- Merge all the tasks into a new coroutine using `asyncio.gather()`
- Run the scan using `asyncio.run()`

```
cpython-book-samples▶33▶portscanner_async.py
```

```
import time
import asyncio

timeout = 1.0

async def check_port(host: str, port: int, results: list):
    try:
        future = asyncio.open_connection(host=host, port=port)
        r, w = await asyncio.wait_for(future, timeout=timeout)
        results.append(port)
        w.close()
    except asyncio.TimeoutError:
        pass # port is closed, skip-and-continue
```

```
async def scan(start, end, host):
    tasks = []
    results = []
    for port in range(start, end):
        tasks.append(check_port(host, port, results))
    await asyncio.gather(*tasks)
    return results

if __name__ == '__main__':
    start = time.time()
    host = "localhost" # pick a host you own
    results = asyncio.run(scan(80, 100, host))
    for result in results:
        print("Port {} is open".format(result))
    print("Completed scan in {} seconds".format(time.time() - start))
```

Finally, this scan completes in just over 1 second:

```
$ python portscanner_async.py
Port 80 is open
Completed scan in 1.0058400630950928 seconds
```

Asynchronous Generators

The concepts you have learned so far, generators and coroutines can be combined into a type - **asynchronous generators**.

If a function is declared with both the `async` keyword and it contains a `yield` statement, it is converted into an `async` generator object when called.

Like generators, `async` generators must be executed by something that understands the protocol. In place of `__next__()`, `async` generators have a method `__anext__()`.

A regular `for` loop would not understand an `async` generator, so instead, the `async for` statement is used.

You can refactor the `check_port()` function into an `async` generator that yields the next open port until it hits the last port, or it has found a specified number of open ports:

```
async def check_ports(host: str, start: int, end: int, max=10):
    found = 0
    for port in range(start, end):
        try:
            future = asyncio.open_connection(host=host, port=port)
            r, w = await asyncio.wait_for(future, timeout=timeout)
            yield port
            found += 1
            w.close()
        if found >= max:
            return
    except asyncio.TimeoutError:
        pass # closed
```

To execute this, use the `async for` statement:

```
async def scan(start, end, host):
    results = []
    async for port in check_ports(host, start, end, max=1):
        results.append(port)
    return results
```

See `cpython-book-samples` ▶ 33 ▶ `portscanner_async_generators.py` for the full example.

Subinterpreters

So far, you have covered:

- Parallel execution with `multiprocessing`
- Concurrent execution with `threads` and `async`

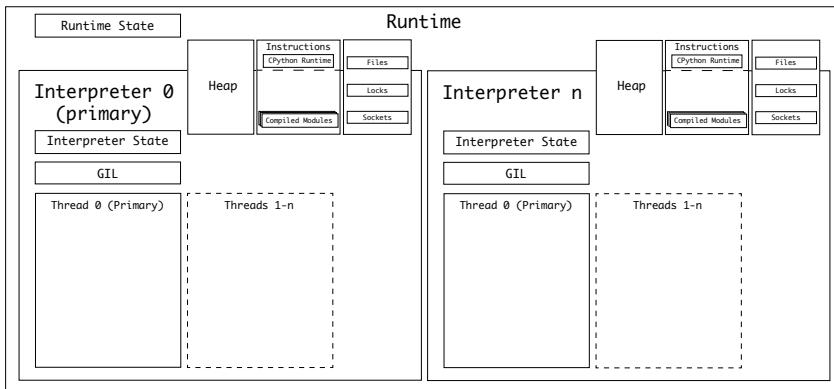
The downside of `multiprocessing` is that the inter-process communication using pipes and queues is slower than shared memory. Also

the overhead to start a new process is significant.

Threading and `async` have small overhead but don't offer truly parallel execution because of the thread-safety guarantees in the GIL.

The fourth option is `subinterpreters`, which have a smaller overhead than `multiprocessing`, and allow a GIL per subinterpreter. After all, it is the Global **Interpreter Lock**.

Within the CPython runtime, there is always 1 interpreter. The interpreter holds the interpreter state, and within an interpreter, you can have 1 or many Python threads. The interpreter is the container for the evaluation loop. The interpreter also manages its own memory, reference counter, and garbage collection. CPython has low-level C APIs for creating interpreters, like the `Py_NewInterpreter()`.



Note

The `subinterpreters` module is still experimental in 3.9.0b1, so the API is subject to change and the implementation is still buggy.

Because Interpreter state contains the memory allocation arena, a collection of all pointers to Python objects (local and global), subinterpreters cannot access the global variables of other interpreters. Simi-

lar to multiprocessing, to share objects between interpreters you must serialize them, or use ctypes, and use a form of IPC (network, disk or shared memory).

Related Source Files

Source files related to subinterpreters are:

File	Purpose
Lib\subinterpreters.c	C implementation of the subinterpreters module
Python\pylifecycle.c	C implementation of the interpreter management API

Example

In the final example application, the actual connection code has to be captured in a string. In 3.9.0b1, subinterpreters can only be executed with a string of code.

To start each of the subinterpreters, a list of threads is started, with a callback to a function, `run()`.

This function will:

- Create a communication channel
- Start a new subinterpreter
- Send it the code to execute
- Receive data over the communication channel
- If the port connection succeeded, add it to the thread-safe queue

cpython-book-samples\33\portscanner_subinterpreters.py

```
import time
import _xxsubinterpreters as subinterpreters
from threading import Thread
```

```
import textwrap as tw
from queue import Queue

timeout = 1 # in seconds.

def run(host: str, port: int, results: Queue):

    # Create a communication channel
    channel_id = subinterpreters.channel_create()
    interp_id = subinterpreters.create()
    subinterpreters.run_string(
        interp_id,
        tw.dedent(
"""
import socket; import _xxsubinterpreters as subinterpreters
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.settimeout(timeout)
result = sock.connect_ex((host, port))
subinterpreters.channel_send(channel_id, result)
sock.close()
"""),
        shared=dict(
            channel_id=channel_id,
            host=host,
            port=port,
            timeout=timeout
        )
    )
    output = subinterpreters.channel_recv(channel_id)
    subinterpreters.channel_release(channel_id)
    if output == 0:
        results.put(port)

if __name__ == '__main__':
    start = time.time()
    host = "127.0.0.1" # pick a host you own
    threads = []
    results = Queue()
    for port in range(80, 100):
        t = Thread(target=run, args=(host, port, results))
```

```
t.start()
threads.append(t)
for t in threads:
    t.join()
while not results.empty():
    print("Port {0} is open".format(results.get()))
print("Completed scan in {0} seconds".format(time.time() - start))
```

Because of the reduced overheads compared with multiprocessing, this example should execute 30-40% faster and with fewer memory resources:

```
$ python portscanner_subinterpreters.py
Port 80 is open
Completed scan in 1.3474230766296387 seconds
```

Conclusion

Congratulations on getting through the biggest chapter in the book! You've covered a lot of ground. Let us recap some of the concepts and their applications.

For truly **parallel execution**, you need multiple CPUs or cores. You also need to use either **multiprocessing** or **subinterpreters** packages so that the Python interpreter can be executed in parallel. Remember that startup time is significant, and each interpreter has a big memory overhead. If the tasks that you want to execute are short-lived, use a pool of workers and a queue of tasks.

If you have multiple IO-bound tasks and want them to run **concurrently**, you should use multithreading, or use coroutines with the **asyncio** package.

All four of these approaches require an understanding of how to safely and efficiently transfer data between processes or threads. The best way to reinforce what you've learned is to look at an application you've written and seen how it can be refactored to leverage these techniques.

[Leave feedback on this section »](#)

Objects and Types

CPython comes with a collection of basic types like strings, lists, tuples, dictionaries, and objects.

All of these types are built-in. You don't need to import any libraries, even from the standard library.

For example, to create a new list, you can call:

```
lst = list()
```

Or, you can use square brackets:

```
lst = []
```

Strings can be instantiated from a string-literal by using either double or single quotes. You explored the grammar definitions in the chapter “The Python Language and Grammar” that cause the compiler to interpret double quotes as a string literal.

All types in Python inherit from `object`, a built-in base type. Even strings, tuples, and lists inherit from `object`.

In `Objects/object.c`, the base implementation of `object` type is written as pure C code. There are some concrete implementations of basic logic, like shallow comparisons.

A simple way to think of a Python object is consisting of 2 things:

1. The core data model, with pointers to compiled functions

2. A dictionary with any custom attributes and methods

Much of the base object API is declared in `Objects/obect.c`, like the implementation of the built-in `repr()` function, `PyObject_Repr`. You will also find `PyObject_Hash()` and other APIs.

All of these functions can be overridden in a custom object by implementing “dunder” methods on a Python object. For example:

```
class MyObject(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def __repr__(self):
        return "<{0} id={1}>".format(self.name, self.id)
```

All of these built-in functions are called the [Python Data Model](#). Not all methods in a Python object are part of the Data Model, so that a Python object can contain attributes (either class or instance attributes) and methods.

See Also

One of the great resources for the Python Data Model is “[Fluent Python](#)” by Luciano Ramalho.

Examples in This Chapter

Throughout this chapter, each type explanation will come with an example. In the example, you will implement the *almost-equal* operator, that was specified in earlier chapters.

If you haven’t yet implemented the changes in the Grammar and Compiler chapters, they will be required to implement the examples.

Builtin Types

The core data model is defined in the `PyTypeObject`, and the functions are defined in:

Each of the source file will have a corresponding header in `Include`. For example, `Objects/rangeobject.c` has a header file `Include/rangeobject.h`.

Source File	Type
<code>Objects/object.c</code>	Built in methods and base object
<code>Objects/boolobject.c</code>	bool type
<code>Objects/bytarrayobject.c</code>	byte[] type
<code>Objects/bytesobjects.c</code>	bytes type
<code>Objects/cellobject.c</code>	cell type
<code>Objects/classobject.c</code>	Abstract class type, used in meta-programming
<code>Objects/codeobject.c</code>	Built-in code object type
<code>Objects/complexobject.c</code>	Complex numeric type
<code>Objects/iterobject.c</code>	An iterator
<code>Objects/listobject.c</code>	list type
<code>Objects/longobject.c</code>	long numeric type
<code>Objects/memoryobject.c</code>	Base memory type
<code>Objects/methodobject.c</code>	Class method type
<code>Objects/moduleobject.c</code>	Module type
<code>Objects/namespaceobject.c</code>	Namespace type
<code>Objects/odictobject.c</code>	Ordered dictionary type
<code>Objects/rangeobject.c</code>	Range generator
<code>Objects/setobject.c</code>	set type
<code>Objects/sliceobject.c</code>	Slice reference type
<code>Objects/structseq.c</code>	struct.Struct type
<code>Objects/tupleobject.c</code>	tuple type
<code>Objects/typeobject.c</code>	type type
<code>Objects/unicodeobject.c</code>	str type
<code>Objects/weakrefobject.c</code>	weakref type

You will explore some of those types in this chapter.

Object and Variable Object Types

Because C is not object-oriented [like Python](#), objects in C don't inherit from one another. `PyObject` is the initial data segment for every Python object and `PyObject *` represents a pointer to it.

When defining Python types, the `typedef` uses one of two macros:

- `PyObject_HEAD` (`PyObject`) for a simple type
- `PyObject_VAR_HEAD` (`PyVarObject`) for a container type

The simple type `PyObject` has the fields:

Field	Type	Purpose
<code>ob_refcnt</code>	<code>Py_ssize_t</code>	Instance reference counter
<code>ob_type</code>	<code>_typeobject*</code>	The object type

For example, the `cellobj` declares 1 additional field, `ob_ref`, and the base fields:

```
typedef struct {
    PyObject_HEAD
    PyObject *ob_ref;      /* Content of the cell or NULL when empty */
} PyCellObject;
```

The variable type, `PyVarObject` extends the `PyObject` type and also has the fields:

Field	Type	Purpose
<code>ob_base</code>	<code>PyObject</code>	The base type
<code>ob_size</code>	<code>Py_ssize_t</code>	Number of items it contains

For example, the `int` type, `PyLongObject`, has the declaration:

```
struct _longobject {
    PyObject_VAR_HEAD
```

```
    digit ob_digit[1];
}; /* PyLongObject */
```

The type Type

In Python, objects have a property `ob_type`, you can get the value of this property using the builtin function `type()`:

```
>>> t = type("hello")
>>> t
<class 'str'>
```

The result from `type()` is an instance of a `PyTypeObject`:

```
>>> type(t)
<class 'type'>
```

Type objects are used to define the implementation of abstract base classes.

For example, objects always have the `__repr__()` method implemented:

```
>>> class example:
...     x = 1
>>> i = example()
>>> repr(i)
'<__main__.example object at 0x10b418100>'
```

The implementation of the `__repr__()` method is always at the same address in the type definition of any object. This position is known as a **type slot**.

Type Slots

All of the type slots are defined in `Include`▸`cpython`▸`object.h`.

Each type slot has a property name and a function signature. The `__repr__()` function for example is called `tp_repr` and has a signature `reprfunc`:

```

struct PyTypeObject
---

typedef struct _typeobject {
    ...
    reprfunc tp_repr;
    ...
} PyTypeObject;

```

The signature `reprfunc` is defined in `Include\cpython\object.h` as having a single argument of `PyObject*` (`self`):

```
typedef PyObject *(*reprfunc)(PyObject *);
```

As an example, the `cell` object implements the `tp_repr` slot with the function `cell_repr`:

```

PyTypeObject PyCell_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "cell",
    sizeof(PyCellObject),
    0,
    (destructor)cell_dealloc,           /* tp_dealloc */
    0,                                /* tp_vectorcall_offset */
    0,                                /* tp_getattr */
    0,                                /* tp_setattr */
    0,                                /* tp_as_async */
    (reprfunc)cell_repr,               /* tp_repr */
    ...
};

```

Beyond the basic `PyTypeObject` type slots, denoted with the `tp_` prefix, there are other type slot definitions:

- `PyNumberMethods` denoted with the prefix `nb_`
- `PySequenceMethods` denoted with the prefix `sq_`
- `PyMappingMethods` denoted with the prefix `mp_`
- `PyAsyncMethods` denoted with the prefix `am_`
- `PyBufferProcs` denoted with the prefix `bf_`

All type slots are given a unique number, defined in `Include/tp_typeslots.h`.

When referring to, or fetching a type slot on an object, use these constants.

For example, `tp_repr` has a constant position of 66, and the constant `Py_tp_repr` always matches the type slot position. These constants are useful when checking if an object implements a particular type slot function.

Working with Types in C

Within C extension modules and the core CPython code, you will be frequently working with the `PyObject*` type.

As an example, if you run `x[n]` on a subscriptable object like a list, or string, it will call `PyObject_GetItem()` which looks at the object `x` to determine how to subscript it:

Objects/abstract.c line 146

```
PyObject *
PyObject_GetItem(PyObject *o, PyObject *key)
{
    PyMappingMethods *m;
    PySequenceMethods *ms;
    ...
}
```

The `PyObject_GetItem()` function serves both mapping types (like dictionaries) as well as sequence types (like lists and tuples).

If the instance, `o` has sequence methods, then `o->ob_type->tp_as_sequence` will evaluate to true, also if the instance, `o`, has a `sq_item` slot function defined, it is assumed that it has correctly implemented the sequence protocol.

The value of `key` is evaluated to check that it is an integer, and the item is requested from the sequence object using the `PySequence_GetItem()`

function:

```
ms = o->ob_type->tp_as_sequence;
if (ms && ms->sq_item) {
    if (PyIndex_Check(key)) {
        Py_ssize_t key_value;
        key_value = PyNumber_AsSsize_t(key, PyExc_IndexError);
        if (key_value == -1 && PyErr_Occurred())
            return NULL;
        return PySequence_GetItem(o, key_value);
    }
    else {
        return type_error("sequence index must "
                           "be integer, not '%.200s'", key);
    }
}
```

Type Property Dictionaries

Python supports defining new types with the `class` keyword. User defined types are created by `type_new()` in the `type` object module.

User defined types will have a property dictionary, accessed by `__dict__()`. Whenever a property is accessed on a custom class, the default `__getattr__` implementation

looks in this property dictionary. Class methods, instance methods, class properties and instance properties are located in this dictionary.

The `PyObject_GenericGetDict()` function implements the logic to fetch the dictionary instance for a given object. The `PyObject_GetAttr()` function implements the default `__getattr__()` implementation and `PyObject_SetAttr()` implements `__setattr__()`.

See Also

There are many layers to custom types, that have been extensively documented in many Python books.

I could fill an entire book on metaclasses, but have decided to stick to the implementation. If you want to learn more, check out [John Sturtz' article on metaprogramming](#).

Bool and Long Integer Type

The `bool` type is the most straightforward implementation of the built-in types. It inherits from `long` and has the predefined constants, `Py_True` and `Py_False`. These constants are **immutable** instances, created on the instantiation of the Python interpreter.

Inside `Objects/boolobject.c`, you can see the helper function to create a `bool` instance from a number:

`Objects/boolobject.c` line 28

```
PyObject *PyBool_FromLong(long ok)
{
    PyObject *result;

    if (ok)
        result = Py_True;
    else
        result = Py_False;
    Py_INCREF(result);
    return result;
}
```

This function uses the C evaluation of a numeric type to assign `Py_True` or `Py_False` to a result and increment the reference counters.

The numeric functions for `and`, `xor`, and `or` are implemented, but addition, subtraction, and division are dereferenced from the base `long` type since it would make no sense to divide two boolean values.

The implementation of `and` for a `bool` value first checks if `a` and `b` are booleans. If they aren't, they are cast as numbers, and the `and` operation is run on the two numbers:

Objects▶boolobject.c line 61

```
static PyObject *
bool_and(PyObject *a, PyObject *b)
{
    if (!PyBool_Check(a) || !PyBool_Check(b))
        return PyLong_Type.tp_as_number->nb_and(a, b);
    return PyBool_FromLong((a == Py_True) & (b == Py_True));
}
```

Long Type

The `long` type is a bit more complex than `bool`. In the transition from Python 2 to 3, CPython dropped support for the `int` type and instead used the `long` type as the primary integer type. Python's `long` type is quite special in that it can store a variable-length number. The maximum length is set in the compiled binary.

The data structure of a Python `long` consists of the `PyObject` variable header and a list of digits. The list of digits, `ob_digit` is initially set to have one digit, but it later expanded to a longer length when initialized:

Include▶longintrepr.h line 85

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};
```

For example, the number `1` would have the `ob_digits` `[1]`, and `24601` would have the `ob_digits` `[2, 4, 6, 0, 1]`.

Memory is allocated to a new `long` through `_PyLong_New()`. This function takes a fixed length and makes sure it is smaller than `MAX_LONG_DIGITS`.

Then it reallocates the memory for `ob_digit` to match the length.

To convert a C `long` type to a Python `long` type, the C `long` is converted to a list of digits, the memory for the Python `long` is assigned, and then each of the digits is set. The `long` object is initialized with `ob_digit` already being at a length of 1 if the number is less than 10 (1 digit). Then, the value is set without the memory being allocated:

Objects ► `longobject.c` line 297

```
PyObject *
PyLong_FromLong(long ival)
{
    PyLongObject *v;
    unsigned long abs_ival;
    unsigned long t; /* unsigned so >> doesn't propagate sign bit */
    int ndigits = 0;
    int sign;

    CHECK_SMALL_INT(ival);
    ...

    /* Fast path for single-digit ints */
    if (!(abs_ival >> PyLong_SHIFT)) {
        v = _PyLong_New(1);
        if (v) {
            Py_SIZE(v) = sign;
            v->ob_digit[0] = Py_SAFE_DOWNCASE(
                abs_ival, unsigned long, digit);
        }
        return (PyObject*)v;
    }

    ...

    /* Larger numbers: loop to determine number of digits */
    t = abs_ival;
    while (t) {
        ++ndigits;
        t >>= PyLong_SHIFT;
    }
}
```

```

v = _PyLong_New(ndigits);
if (v != NULL) {
    digit *p = v->ob_digit;
    Py_SIZE(v) = ndigits*sign;
    t = abs_ival;
    while (t) {
        *p++ = Py_SAFE_DOWNCAST(
            t & PyLong_MASK, unsigned long, digit);
        t >>= PyLong_SHIFT;
    }
}
return (PyObject *)v;
}

```

To convert a double-point floating point to a Python `long`, `PyLong_FromDouble()` does the math for you.

The remainder of the implementation functions in `Objects/longobject.c` have utilities, such as converting a Unicode string into a number with `PyLong_FromUnicodeObject()`.

Example

The rich-comparison type slot for `long` is set to the `long_richcompare`. This function wraps `long_compare`:

`Objects/longobject.c` line 3031

```

static PyObject *
long_richcompare(PyObject *self, PyObject *other, int op)
{
    Py_ssize_t result;
    CHECK_BINOP(self, other);
    if (self == other)
        result = 0;
    else
        result = long_compare((PyLongObject*)self, (PyLongObject*)other);
    Py_RETURN_RICHCOMPARE(result, 0, op);
}

```

The `long_compare` function will first check whether the length (number of digits) of the two variables `a` and `b`. If the lengths are the same, it then loops through each digit to see if they are equal to each other.

`long_compare()` returns:

- A negative number when $a < b$
- 0 when $a < b$
- A positive number when $a > b$

For example, when you execute `1 == 5`, the result is `-4`. `5 == 1`, the result is `4`.

You can implement the following code block before the `Py_RETURN_RICHCOMPARE` macro to return `True` when the absolute value of result is ≤ 1 using the macro `Py_ABS()`, which returns the absolute value of a signed integer:

```
if (op == Py_EQE) {  
    if (Py_ABS(result) <= 1)  
        Py_RETURN_TRUE;  
    else  
        Py_RETURN_FALSE;  
}  
Py_RETURN_RICHCOMPARE(result, 0, op);  
}
```

After recompiling Python, you should see the effect of the change:

```
>>> 2 == 1  
False  
>>> 2 ~== 1  
True  
>>> 2 ~== 10  
False
```

Unicode String Type

Python Unicode strings are complicated. Cross-platform Unicode types in any platform are complicated.

The cause of this complexity is the number of encodings that are on offer, and the different default configurations on the platforms that Python supports.

The Python 2 string type was stored in C using the `char` type. The 1-byte `char` type sufficiently stores any of the ASCII (American Standard Code for Information Interchange) characters and has been used in computer programming since the 1970s.

ASCII does not support the 1000's of languages and character sets that are in use across the world. Also, there are extended glyph character sets like emojis, which it cannot support.

A standard system of coding and a database of characters was created, known as the Unicode Standard. The modern Unicode Standard includes characters for all written languages, as well as extended glyphs and characters. The **Unicode Character Database** (UCD) contains 137,929 at version 12.1 (compared with the 128 in ASCII).

The Unicode standard defines these characters in a character table called the **Universal Character Set (UCS)**. Each character has a unique identifier known as a **code point**.

There are then many **encodings** that use the Unicode Standard and convert the code-point into a binary value.

Python Unicode strings support three lengths of encodings:

- 1-byte (8-bit)
- 2-byte (16-bit)
- 4-byte (32-bit)

These variable-length encodings are referred to within the implemen-

tation as:

- 1-byte `Py_UCS1`, stored as 8-bit unsigned int type, `uint8_t`
- 2-byte `Py_UCS2`, stored as 16-bit unsigned int type, `uint16_t`
- 4-byte `Py_UCS4`, stored as 32-bit unsigned int type, `uint32_t`

Related Source Files

Source files related to strings are:

File	Purpose
Include <code>>unicodeobject.h</code>	Unicode String Object definition
Include <code>>cpython >unicodeobject.h</code>	Unicode String Object definition
Objects <code>>unicodeobject.c</code>	Unicode String Object implementation
Lib <code>>encodings</code>	Encodings package containing all the possible encodings
Lib <code>>codecs.py</code>	Codecs module
Modules <code>>_codecsmodule.c </code>	Codecs module C extensions, implements OS-specific encodings
Modules <code>>_codecs </code>	Codec implementations for a range of alternative encodings

Processing Unicode Code Points

CPython does not contain a copy of the UCD, nor does it have to update whenever scripts and characters are added to the Unicode standard. Unicode Strings in CPython only have to care about the encodings, the Operating System has the task of representing the code points in the correct scripts.

The Unicode standard includes the UCD and is updated regularly with new scripts, new Emojis, and new characters.

Operating Systems take on these updates to Unicode and update their software via a patch. These patches include the new UCD code-points and support the various Unicode encodings. The UCD is split into sections called **code blocks**.

The Unicode Code charts are published on the [Unicode Website](#).

Another point of support for Unicode is the Web Browser. Web Browsers decode HTML binary data in the encoding marked HTTP encoding headers. If you are working with CPython as a web server, then your Unicode encodings must match the HTTP headers being sent to your users.

UTF8 vs UTF16

Some common encodings are:

- **UTF8**, an 8-bit character encoding that supports all possible characters in the UCD with either a 1-4 byte code point
- **UTF16**, a 16-bit character encoding, similar to UTF8, but not compatible with 7 or 8-bit encodings like ASCII

UTF8 is the most commonly used Unicode encoding.

In all Unicode encodings, the code points can be represented using a hexadecimal shorthand:

- U+00F7 for the division character '÷'
- U+0107 for the Latin Small Letter C with acute 'ć'

In Python, Unicode code points can be encoded directly into the code using the `u` escape symbol and the hexadecimal value of the code point:

```
>>> print("u0107")
ć
```

CPython does not attempt to pad this data, so if you tried `u107`, it would give the following exception:

```
print("u107")
File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode
bytes in position 0-4: truncated uXXXX escape
```

Both XML and HTML support unicode code points with a special escape character `&#val;`, where `val` is the decimal value of the code point. If you need to encode Unicode code points into XML or HTML, you can use the `xmlcharrefreplace` error-handler in the `.encode()` method:

```
>>> "u0107".encode('ascii', 'xmlcharrefreplace')
b'&#263;'
```

The output will contain HTML/XML-escaped code-points. All modern browsers will decode this escape sequence into the correct character.

ASCII Compatibility

If you are working with ASCII-encoded text, it is important to understand the difference between UTF7/8 and UTF16. UTF8 has a major benefit of being compatible with ASCII encoded text. ASCII encoding is a 7-bit encoding.

The first 128 code points on the Unicode Standard represent the existing 128 characters of the ASCII standard. For example, the Latin letter 'a' is the 97th character in ASCII and the 97th character in Unicode. Decimal 97 is equivalent to 61 in hexadecimal, so the Unicode code point is `U+0061`.

On the REPL, if you create the binary code for the letter 'a':

```
>>> letter_a = b'a'
>>> letter_a.decode('utf8')
'a'
```

This can correctly be decoded into UTF8.

UTF16 works with 2-4 byte code points. The 1-byte representation of the letter 'a' will not decode:

```
>>> letter_a.decode('utf16')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
UnicodeDecodeError: 'utf-16-le' codec can't decode
byte 0x61 in position 0: truncated data
```

This is important to note when you are selecting an encoding mechanism. UTF8 is a safer option if you need to import ASCII encoded data.

Wide Character Type

When handling Unicode string input in an unknown encoding within the CPython source code, the `wchar_t` C type will be used. `wchar_t` is the C standard for a wide-character string and is sufficient to store Unicode strings in memory. After [PEP 393](#), the `wchar_t` type was selected as the Unicode storage format.

The Unicode string object provides a utility function, `PyUnicode_FromWideChar()`, that will convert a `wchar_t` constant to a string object. For example, the `pymain_run_command()`, used by `python -c` converts the `-c` argument into a Unicode string:

Modules \rightarrow `main.c` line 226

```
static int
pymain_run_command(wchar_t *command, PyCompilerFlags *cf)
{
    PyObject *unicode, *bytes;
    int ret;

    unicode = PyUnicode_FromWideChar(command, -1);
```

Byte Order Markers

When decoding an input, like a file, CPython can detect the byte-order from a byte-order-marker (BOM). BOMs are a special character that appears at the beginning of a Unicode byte stream. They tell the receiver which byte-order the data is stored in. Different computer systems can encode with different byte-orders. If the wrong byte-order is used, even with the right encoding, the data will be garbled.

A **big-endian** ordering places the most significant byte first. A **little-endian** ordering places the least significant byte first.

The UTF8 specification does support a BOM, but it has no effect. The UTF8 BOM can appear at the beginning of a UTF8-encoded data sequence, represented as `b'\xef\xbb\xbf'`, and will indicate to CPython that the data stream is most-likely UTF8. UTF16 and UTF32 support little and big-endian BOMs.

The default byte-order in CPython is set by the `sys.byteorder` global value:

```
>>> import sys; print(sys.byteorder)
little
```

The Encodings Package

The encodings package in `Lib\encodings` comes with over 100 builtin supported encodings for CPython.

Whenever the `.encode()` or `.decode()` method is called on a string or byte string, the encoding is looked up from this package.

Each encoding is defined as a separate module, e.g., `ISO2022_JP`, a widely used encoding for Japanese email systems, is declared in `Lib\encodings\iso2022_jp.py`.

Every encoding module will define a function `getregentry()` and register:

- Its unique name
- Its encode and decode functions from a codec module
- Its incremental encoder and decoder classes
- Its stream reader and stream writer classes

Many of the encoding modules share the same codecs either from the `codecs` module, or the `_multibytecodec` module. Some encoding modules use a separate codec module in C, from `Modules_codecs`.

For example, the `ISO2022_JP` encoding module imports a C extension module, `_codecs_iso2022`, from `Modules`▶`_codecs`▶`_codecs_iso2022.c`:

```
import _codecs_iso2022, codecs
import _multibytecodec as mbc

codec = _codecs_iso2022.getcodec('iso2022_jp')

class Codec(codecs.Codec):
    encode = codec.encode
    decode = codec.decode

class IncrementalEncoder(mbc.MultibyteIncrementalEncoder,
                        codecs.IncrementalEncoder):
    codec = codec

class IncrementalDecoder(mbc.MultibyteIncrementalDecoder,
                        codecs.IncrementalDecoder):
    codec = codec
```

The `encodings` package also has a module, `Lib`▶`encodings`▶`aliases.py`, containing a dictionary, `aliases`. This dictionary is used to map encodings in the registry by alternative names. For example, `utf8`, `utf-8` and `u8` are all aliases of the `utf_8` encoding.

The Codecs Module

The `codecs` module handles the translation of data with a specific encoding.

The encode or decode function of a particular encoding can be fetched using the `getencoder()` and `getdecoder()` functions respectively:

```
>>> iso2022_jp_encoder = codecs.getencoder('iso2022_jp')
>>> iso2022_jp_encoder('u3072u3068') # hi-to
(b'x1b$B$R$Hx1b(B', 2)
```

The encode function will return the binary result and the number of bytes in the output as a tuple.

The `codecs` module also implements the builtin function `open()` for opening file handles from the operating system.

Codec Implementations

In the `Unicode Object (Objects/unicodeobject.c)` implementation are the encoding and decoding methods for:

Codec	Encoder / Decoder
ascii	<code>PyUnicode_EncodeASCII()</code> / <code>PyUnicode_DecodeASCII()</code>
latin1	<code>PyUnicode_EncodeLatin1()</code> / <code>PyUnicode_DecodeLatin1()</code>
UTF7	<code>PyUnicode_EncodeUTF7()</code> / <code>PyUnicode_DecodeUTF7()</code>
UTF8	<code>PyUnicode_EncodeUTF8()</code> / <code>PyUnicode_DecodeUTF8()</code>
UTF16	<code>PyUnicode_EncodeUTF16()</code> / <code>PyUnicode_DecodeUTF16()</code>
UTF32	<code>PyUnicode_EncodeUTF32()</code> / <code>PyUnicode_DecodeUTF32()</code>
unicode_escape	<code>PyUnicode_EncodeUnicodeEscape()</code> / <code>PyUnicode_DecodeUnicodeEscape()</code>
raw_unicode_ escape	<code>PyUnicode_EncodeRawUnicodeEscape()</code> / <code>PyUnicode_DecodeRawUnicodeEscape()</code>

The implementation of the other encodings is within `Modules/_codecs` to avoid cluttering the main unicode string object implementation.

The `unicode_escape` and `raw_unicode_escape` codecs are internal to CPython.

Internal Codecs

CPython comes with a number of internal encodings. These are unique to CPython and useful for some of the standard library functions, and when working with producing source code.

These text encodings can be used with any text input and output:

Codec	Purpose
idna	Implements RFC 3490
mbcs	(Windows only): Encode according to the ANSI codepage
raw_unicode_escape	Convert to a string for raw literal in Python source code

Codec	Purpose
string_escape	Convert to a string literal for Python source code
undefined	Try default system encoding
unicode_escape	Convert to Unicode literal for Python source code
unicode_internal	Return the internal CPython representation

These binary-only encodings need to be used with `codecs.encode()`/
`codecs.decode()` with byte string inputs, e.g., :

```
>>> codecs.encode(b'hello world', 'base64')
b'aGVsbG8gd29ybGQ=n'
```

Codec	Aliases	Purpose
base64_codec	base64, base-64	Convert to MIME base64
bz2_codec	bz2	Compress the string using bz2
hex_codec	hex	Convert to hexadecimal representation, with two digits per byte
quopri_codec	quoted-printable	Convert operand to MIME quoted printable
rot_13	rot13	Returns the Caesar-cypher encryption (position 13)
uu_codec	uu	Convert using uuencode
zlib_codec	zip, zlib	Compress using gzip

Example

The `tp_richcompare` type slot is allocated to the `PyUnicode_RichCompare()` function in the `PyUnicode_Type`. This function does the comparison of strings and can be adapted to the `~=` operator.

The behaviour you will implement is a case-insensitive comparison of the two strings.

First, add an additional case statement to check when the left and right strings have binary equivalence.

Objects ▶ `unicodeobject.c` line 11350

```

PyObject *
PyUnicode_RichCompare(PyObject *left, PyObject *right, int op)
{
    ...
    if (left == right) {
        switch (op) {
            case Py_EQ:
            case Py_LE:
            >>>    case Py_ALE:
            case Py_GE:
                /* a string is equal to itself */
                Py_RETURN_TRUE;

```

Then add a new `else if` block to handle the `Py_ALE` operator. This will:

1. Convert the left string to a new upper-case string
2. Convert the right string to a new upper-case string
3. Compare the two
4. Dereference both of the temporary strings so they get deallocated
5. Return the result

Your code should look like this:

```

else if (op == Py_EQ || op == Py_NE) {
    ...
}

/* Add these lines */

else if (op == Py_ALE){
    PyObject* upper_left = case_operation(left, do_upper);
    PyObject* upper_right = case_operation(right, do_upper);
    result = unicode_compare_eq(upper_left, upper_right);
    Py_DECREF(upper_left);
    Py_DECREF(upper_right);
    return PyBool_FromLong(result);
}

```

After recompiling, your case-insensitive string matching should give the following results on the REPL:

```
>>> "hello" ~= "HEllo"  
True  
>>> "hello?" ~= "hello"  
False
```

Dictionary Type

Dictionaries are a fast and flexible mapping type. They are used by developers to store and map data, as well as by Python objects to store properties and methods.

Python dictionaries are also used for local and global variables, for keyword arguments and many other use cases.

Python dictionaries are **compact**, meaning the hash table only stores mapped values.

The hashing algorithm that is part of all immutable builtin types is fast, and what gives Python dictionaries their speed.

Hashing

All immutable builtin types provide a hashing function. This is defined in the `tp_hash` type slot, or using the `__hash__()` magic-method for custom types. Hash values are the same size as a pointer (64-bit for 64-bit systems, 32 for 32-bit systems), but do not represent the memory address of their values.

The resulting hash for any Python Object should not change during it's lifecycle. Hashes for two immutable instances with identical values should be equal:

```
>>> "hello".__hash__() == ("hel" + "lo").__hash__()  
True
```

There should be no hash collisions, two objects with different values should not produce the same hash.

Some hashes are simple, like Python longs:

```
>>> (401).__hash__()  
401
```

Long hashes get more complex for a longer value:

```
>>> (401123124389798989898).__hash__()  
2212283795829936375
```

Many of the builtin types use the `Python>pyhash.c` module, which provides a hashing helper function for:

- Bytes `_Py_HashBytes(const void*, Py_ssize_t)`
- Double `_Py_HashDouble(double)`
- Pointers `_Py_HashPointer(void*)`

Unicode strings for example, use `_Py_HashBytes()` to hash the byte data of the string:

```
>>> ("hello").__hash__()  
4894421526362833592
```

Custom classes can define a hashing function by implementing `__hash__()`. Instead of implementing a custom hash, custom classes should use a **unique** property. Make sure it is immutable by making it a read-only property, then use hash it using the builtin `hash()` function:

```
class User:  
    def __init__(self, id: int, name: str, address: str):  
        self._id = id  
  
    def __hash__(self):  
        return hash(self._id)  
  
    @property  
    def id(self):  
        return self._id
```

Instances of this class can now be hashed:

```
>>> bob = User(123884, "Bob Smith", "Townsville, QLD")
>>> hash(bob)
123884
```

This instance can now be used as a dictionary key:

```
>>> sally = User(123823, "Sally Smith", "Cairns, QLD")
>>> near_reef = {bob: False, sally: True}
>>> near_reef[bob]
False
```

Sets will reduce duplicate hashes of this instance:

```
>>> {bob, bob}
{<__main__.User object at 0x10df244b0>}
```

Related Source Files

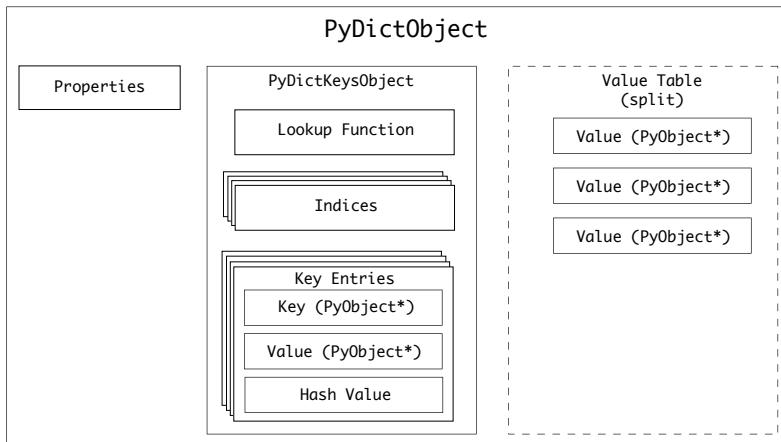
Source files related to dictionaries are:

File	Purpose
Include\dictobject.h	Dictionary Object API definition
Include\cpython\dictobject.h	Dictionary Object types definition
Objects\dictobject.c	Dictionary Object implementation
Objects\dict-common.h	Definition of key entry, and key objects
Python\pyhash.c	Internal hashing algorithm

Dictionary Structure

Dictionary objects, `PyDictObject` are comprised of:

1. The Dictionary Object, `PyDictObject`, containing the size, a version tag, the keys and values
2. A Dictionary Keys Object, containing the keys and hash values of all entries



The PyDictObject has the properties:

Field	Type	Purpose
<code>ma_used</code>	<code>Py_ssize_t</code>	Number of items in the dictionary
<code>ma_</code>	<code>uint64_t</code>	Version number of the dictionary
<code>version_tag</code>		
<code>ma_keys</code>	<code>PyDictKeysObject</code>	Dictionary Key Table Object
	*	
<code>ma_values</code>	<code>PyObject **</code>	Optional value array (see note)

Note

Dictionaries can have two states- split or combined. When dictionaries are combined, the pointers to the dictionary values are stored in the keys object.

When the dictionary is split, the values are stored in an extra property, `ma_values`, as a value table of `PyObject*`.

The dictionary key table, PyDictKeysObject, contains:

Field	Type	Purpose
<code>dk_refcnt</code>	<code>Py_ssize_t</code>	Reference counter

Field	Type	Purpose
dk_size	Py_ssize_t	The size of the hash table
dk_lookup	dict_lookup_- func	The lookup function (See next section)
dk_usable	Py_ssize_t	The number of usable entries in the entry table, when 0, dictionary is resized
dk_- nentries	Py_ssize_t	The number of used entries in the entry table
dk_- indices	char[]	Hash table and mapping to dk_entries
dk_- entries	PyDictKeyEntry[]	Allocated array of dictionary key entries

A dictionary key entry, `PyDictKeyEntry` contains:

Field	Type	Purpose
me_hash	Py_ssize_t	Cached hash code of <code>me_key</code>
me_key	PyObject*	Pointer to the key object
me_value	PyObject*	Pointer to the value object (if combined)

Lookups

For a given key object, there is a generic lookup function `lookdict()`.

Dictionary lookups need to cater for three scenarios:

1. The memory address of the key exists in the key table
2. The hash value of the object exists in the key table
3. The key does not exist in the dictionary

See Also

The lookup function is based on Donald Knuth's famous book, "The art of Computer Programming", chapter 6, section 4 on hashing (ISBN 978-0201896855)

The sequence of the lookup function is:

1. Get the hash value of `ob`
2. Lookup the hash value of `ob` in the dictionary keys and get the index, `ix`
3. If `ix` is empty, return `DKIX_EMPTY` (not found)
4. Get the key entry, `ep` for the given index
5. If the key values match because the object, `ob` is the same pointer at the key value, return the result
6. If the key hashes match because the object, `ob` resolves to the same hash value as `ep->me_mash`, return the result

Note

The `lookupdict()` function is one of few “hot functions” in the CPython source code.

“The `hot` attribute is used to inform the compiler that a function is a hot spot of the compiled program. The function is optimized more aggressively and on many target it is placed into special subsection of the text section so all hot functions appears close together improving locality.”

This is specific to GNUC compilers, but when compiled with PGO, this function is likely to be optimized by the compiler automatically.

Conclusion

Now that you have seen the implementation of some built-in types, you can explore others.

When exploring Python classes, it is important to remember there are built-in types, written in C and classes inheriting from those types, written in Python or C.

Some libraries have types written in C instead of inheriting from the

built-in types. One example is `numpy`, a library for numeric arrays. The `ndarray` type is written in C, is highly efficient and performant.

In the next chapter, you will explore the classes and functions defined in the standard library.

[Leave feedback on this section »](#)

The Standard Library

Python has always come “batteries included.” This statement means that with a standard CPython distribution, there are libraries for working with files, threads, networks, web sites, music, keyboards, screens, text, and a whole manner of utilities.

Some of the batteries that come with CPython are more like AA batteries. They’re useful for everything, like the `collections` module and the `sys` module. Some of them are a bit more obscure, like a small watch battery that you never know when it might come in useful.

There are two types of modules in the CPython standard library:

1. Those written in pure Python that provide a utility
2. Those written in C with Python wrappers

You will explore both types in this chapter.

Python Modules

The modules written in pure Python are all located in the `Lib` directory in the source code. Some of the larger modules have submodules in subfolders, like the `email` module.

An easy module to look at would be the `colorsys` module. It’s only a few hundred lines of Python code. You may not have come across it before. The `colorsys` module has some utility functions for converting color scales.

When you install a Python distribution from source, standard library modules are copied from the `Lib` folder into the distribution folder. This folder is always part of your path when you start Python, so you can `import` the modules without having to worry about where they're located.

For example:

```
>>> import colorsys
>>> colorsys
<module 'colorsys' from '/usr/shared/lib/python3.7/colorsys.py'>

>>> colorsys.rgb_to_hls(255,0,0)
(0.0, 127.5, -1.007905138339921)
```

We can see the source code of `rgb_to_hls()` inside `Lib\colorsys.py`:

```
# HLS: Hue, Luminance, Saturation
# H: position in the spectrum
# L: color lightness
# S: color saturation

def rgb_to_hls(r, g, b):
    maxc = max(r, g, b)
    minc = min(r, g, b)
    # XXX Can optimize (maxc+minc) and (maxc-minc)
    l = (minc+maxc)/2.0
    if minc == maxc:
        return 0.0, l, 0.0
    if l <= 0.5:
        s = (maxc-minc) / (maxc+minc)
    else:
        s = (maxc-minc) / (2.0-maxc-minc)
    rc = (maxc-r) / (maxc-minc)
    gc = (maxc-g) / (maxc-minc)
    bc = (maxc-b) / (maxc-minc)
    if r == maxc:
        h = bc-gc
```

```
elif g == maxc:  
    h = 2.0+rc-bc  
else:  
    h = 4.0+gc-rc  
    h = (h/6.0) % 1.0  
return h, 1, s
```

There's nothing special about this function, it's just standard Python. You'll find similar things with all of the pure Python standard library modules. They're just written in plain Python, well laid out and easy to understand. You may even spot improvements or bugs, so you can make changes to them and contribute it to the Python distribution. You'll cover that toward the end of this book.

Python and C Modules

The remainder of modules are written in C, or a combination of Python and C. The source code for these is in `Lib` for the Python component, and `Modules` for the C component. There are two exceptions to this rule, the `sys` module, found in `Python` ▶ `sysmodule.c` and the `__builtins__` module, found in `Python` ▶ `bltinmodule.c`.

Python will `import *` from `__builtins__` when an interpreter is instantiated, so all of the functions like `print()`, `chr()`, `format()`, etc. are found within `Python` ▶ `bltinmodule.c`.

Because the `sys` module is so specific to the interpreter and the internals of CPython, that is found inside the `Python` directory. It is also marked as an “implementation detail” of CPython and not found in other distributions.

The built-in `print()` function was probably the first thing you learned to do in Python. So what happens when you type `print("hello world!")?`

1. The argument "hello world" was converted from a string constant to a `PyUnicodeObject` by the compiler

2. `builtin_print()` was executed with 1 argument, and NULL `kwnames`
3. The `file` variable is set to `PyId_stdout`, the system's `stdout` handle
4. Each argument is sent to `file`
5. A line break, `\n` is sent to `file`

Python▶bltinmodule.c line 1828

```
static PyObject *
builtin_print(PyObject *self, PyObject *const *args,
             Py_ssize_t nargs, PyObject *kwnames)
{
    ...
    if (file == NULL || file == Py_None) {
        file = _PySys_GetObjectId(&PyId_stdout);
        ...
    }
    ...
    for (i = 0; i < nargs; i++) {
        if (i > 0) {
            if (sep == NULL)
                err = PyFile_WriteString(" ", file);
            else
                err = PyFile_WriteObject(sep, file,
                                         Py_PRINT_RAW);
        if (err)
            return NULL;
        }
        err = PyFile_WriteObject(args[i], file, Py_PRINT_RAW);
        if (err)
            return NULL;
    }

    if (end == NULL)
        err = PyFile_WriteString("\n", file);
    else
        err = PyFile_WriteObject(end, file, Py_PRINT_RAW);
    ...
}
```

```
    Py_RETURN_NONE;  
}
```

The contents of some modules written in C expose operating system functions. Because the CPython source code needs to compile to macOS, Windows, Linux, and other *nix-based operating systems, there are some special cases.

The `time` module is a good example. The way that Windows keeps and stores time in the Operating System is fundamentally different than Linux and macOS. This is one of the reasons why the accuracy of the clock functions differs [between Operating Systems](#).

In `Modules/timemodule.c`, the Operating System time functions for Unix-based systems are imported from `<sys/times.h>`:

```
#ifdef HAVE_SYS_TIMES_H  
#include <sys/times.h>  
#endif  
...  
#ifdef MS_WINDOWS  
#define WIN32_LEAN_AND_MEAN  
#include <windows.h>  
#include "pythread.h"  
#endif /* MS_WINDOWS */  
...
```

Later in the file, `time_process_time_ns()` is defined as a wrapper for `_PyTime_GetProcessTimeWithInfo()`:

```
static PyObject *  
time_process_time_ns(PyObject *self, PyObject *unused)  
{  
    _PyTime_t t;  
    if (_PyTime_GetProcessTimeWithInfo(&t, NULL) < 0) {  
        return NULL;  
    }  
    return _PyTime_AsNanosecondsObject(t);  
}
```

`_PyTime_GetProcessTimeWithInfo()` is implemented multiple different ways in the source code, but only certain parts are compiled into the binary for the module depending on the operating system. Windows systems will call `GetProcessTimes()` and Unix systems will call `clock_gettime()`.

Other modules that have multiple implementations for the same API are [the threading module](#), the file system module, and the networking modules. Because the Operating Systems behave differently, the CPython source code implements the same behavior as best as it can and exposes it using a consistent, abstracted API.

[Leave feedback on this section »](#)

The Test Suite

CPython has a robust and extensive test suite covering the core interpreter, the standard library, the tooling, and distribution for Windows, Linux, and macOS.

The test suite is located in `Lib\test` and written almost entirely in Python.

The full test suite is a Python package, so it can be run using the Python interpreter that you've compiled.

Running the Test Suite on Windows

On Windows use the `rt.bat` script inside the `PCBuild` folder.

For example, to run the “quick” mode against the Debug configuration on an x64 architecture:

```
> cd PCbuild
> rt.bat -q -d -x64

== CPython 3.9.0b1
== Windows-10-10.0.17134-SP0 little-endian
== cwd: C:\repos\cpython\build\test_python_2784
== CPU count: 2
== encodings: locale=cp1252, FS=utf-8
Run tests sequentially
0:00:00 [ 1/420] test_grammar
```

```
0:00:00 [ 2/420] test_OPCODE
0:00:00 [ 3/420] test_dict
0:00:00 [ 4/420] test_builtin
...
...
```

To run the regression test suite against the Release configuration, remove the `-d` flag from the command-line

Running the Test Suite on Linux/macOS

On Linux or macOS run the `test` make target to compile and run the tests:

```
$ make test
== CPython 3.9.0b1
== macOS-10.14.3-x86_64-i386-64bit little-endian
== cwd: /Users/anthonyshaw/cpython/build/test_python_23399
== CPU count: 4
== encodings: locale=UTF-8, FS=utf-8
0:00:00 load avg: 2.14 [ 1/420] test_OPCODE passed
0:00:00 load avg: 2.14 [ 2/420] test_grammar passed
...
...
```

Alternatively, use the `python.exe` compiled binary path with the `test` package:

```
$ ./python.exe -m test
== CPython 3.9.0b1
== macOS-10.14.3-x86_64-i386-64bit little-endian
== cwd: /Users/anthonyshaw/cpython/build/test_python_23399
== CPU count: 4
== encodings: locale=UTF-8, FS=utf-8
0:00:00 load avg: 2.14 [ 1/420] test_OPCODE passed
0:00:00 load avg: 2.14 [ 2/420] test_grammar passed
...
...
```

There are additional make targets for testing:

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

Test Flags

Some tests require certain flags; otherwise they are skipped. For example, many of the IDLE tests require a GUI.

To see a list of test suites in the configuration, use the `--list-tests` flag:

```
$ ./python -m test --list-tests

test_grammar
test_opcodes
test_dict
test_builtin
test_exceptions
...
```

Running Specific Tests

You can run specific tests by providing the test suite as the first argument:

On Linux or macOS:

```
$ ./python -m test test_webbrowser

Run tests sequentially
```

```
0:00:00 load avg: 2.74 [1/1] test_webbrowser

== Tests result: SUCCESS ==

1 test OK.

Total duration: 117 ms
Tests result: SUCCESS
```

On Windows:

```
> rt.bat -q -d -x64 test_webbrowser
```

You can also see a detailed list of tests that were executed with the result using the `-v` argument:

```
$ ./python -m test test_webbrowser -v

== CPython 3.9.0b1
== macOS-10.14.3-x86_64-i386-64bit little-endian
== cwd: /Users/anthonyshaw/cpython/build/test_python_24562
== CPU count: 4
== encodings: locale=UTF-8, FS=utf-8
Run tests sequentially
0:00:00 load avg: 2.36 [1/1] test_webbrowser
test_open (test.test_webbrowser.BackgroundBrowserCommandTest) ... ok
test_register (test.test_webbrowser.BrowserRegistrationTest) ... ok
test_register_default (test.test_webbrowser.BrowserRegistrationTest) ... ok
test_register_preferred (test.test_webbrowser.BrowserRegistrationTest) ... ok
test_open (test.test_webbrowser.ChromeCommandTest) ... ok
test_open_new (test.test_webbrowser.ChromeCommandTest) ... ok
...
test_open_with_autoraise_false (test.test_webbrowser.OperaCommandTest) ... ok
```

```
Ran 34 tests in 0.056s
```

```
OK (skipped=2)
```

```
== Tests result: SUCCESS ==

1 test OK.

Total duration: 134 ms
Tests result: SUCCESS
```

Understanding how to use the test suite and checking the state of the version you have compiled is very important if you wish to make changes to CPython. Before you start making changes, you should run the whole test suite and make sure everything is passing.

Testing Modules

To test C extension or Python modules, they are imported and tested using the `unittest` module. Tests are assembled by module or package.

For example, the Python Unicode string type has tests in `Lib\test\test_unicode.py`. The `asyncio` package has a test package in `Lib\test\test_asyncio`.

See Also

If you're new to the `unittest` module or testing in Python, check out my [Getting Started With Testing in Python](#) article on [realpython.com](#)

```
class UnicodeTest(string_tests.CommonTest,
    string_tests.MixinStrUnicodeUserStringTest,
    string_tests.MixinStrUnicodeTest,
    unittest.TestCase):

...
    def test_casifold(self):
        self.assertEqual('hello'.casifold(), 'hello')
        self.assertEqual('hELLo'.casifold(), 'hello')
        self.assertEqual('ß'.casifold(), 'ss')
        self.assertEqual('fi'.casifold(), 'fi')
```

You can extend the almost-equal operator that you implemented for Python Unicode strings in earlier chapters by adding a new test method inside the `UnicodeTest` class:

```
def test_almost_equals(self):
    self.assertTrue('hello' ~= 'hello')
    self.assertTrue('hELlo' ~= 'hello')
    self.assertFalse('hELlo!' ~= 'hello')
```

You can run this particular test module on Windows:

```
> rt.bat -q -d -x64 test_unicode
```

Or macOS/Linux:

```
$ ./python -m test test_unicode -v
```

Test Utilities

By importing the `test.support.script_helper` module, you can access some helper functions for testing the Python runtime:

- `assert_python_ok(*args, **env_vars)` executes a Python process with the specified arguments and returns a `(return code, stdout, stderr)` tuple
- `assert_python_failure(*args, **env_vars)` similar to `assert_python_ok()`, but asserts that it fails to execute
- `make_script(script_dir, script_basename, source)` makes a script in `script_dir` with the `script_basename` and the `source`, then returns the `script path`. Useful to combine with `assert_python_ok()` or `assert_python_failure()`

If you want to create a test that is skipped if the module wasn't built, you can use the `test.support.import_module()` utility function. It will raise a `SkipTest` and signal the test runner to skip this test package, for example:

```
import test.support

_multiprocessing = test.support.import_module('_multiprocessing')

# Your tests...
```

Conclusion

The Python regression test suite is full of two decades of tests for strange edge cases, bug fixes, and new features. Outside of this, there is still a large part of the CPython standard library that has little or no testing. If you want to get involved in the CPython project, writing or extending unit tests is a great place to start.

If you're going to modify any part of CPython or add additional functionality, you will need to have written, or extended tests as part of your patch.

[Leave feedback on this section »](#)

Debugging

C₀Python comes with a builtin debugger for debugging Python applications, `pdb`. The `pdb` debugger is excellent for debugging crashes inside a Python application, for writing tests and inspecting local variables.

When it comes to C₀Python, you need a second debugger, one that understands C.

In this chapter, you will learn how to:

- Attach a debugger to the C₀Python interpreter
- Use the debugger to see inside a running C₀Python process

There are two types of debugger, console and visual. Console debuggers (like `pdb`) give you a command prompt and custom commands to explore variables and the stack. Visual debuggers are GUI applications that present the data for you in grids.

The following debuggers are covered in this chapter:

Debugger	Type	Platform
lldb	Console	macOS
gdb	Console	Linux
Visual Studio Debugger	Visual	Windows
CLion Debugger	Visual	Windows, macOS, Linux
VS Code Debugger	Visual	Windows, macOS, Linux

Using the Crash Handler

In C, if an application tries to read or write to an area of memory that it shouldn't be, a segmentation fault is raised. This fault halts the running process immediately to stop it from doing any damage to other applications.

Segmentation faults can also happen when you try to read from memory that contains no data, or an invalid pointer.

If CPython causes a segmentation fault, you get very little information about what happened:

```
[1] 63476 segmentation fault ./python portscanner.py
```

CPython comes with a builtin fault handler. If you start CPython with `-X faulthandler`, or `-X dev`, instead of printing the system segmentation fault message, the fault handler will print the running threads and the Python stack trace to where the fault occurred:

```
Fatal Python error: Segmentation fault
Thread 0x0000000119021dc0 (most recent call first):
  File "/cpython/Lib/threading.py", line 1039 in _wait_for_tstate_lock
  File "/cpython/Lib/threading.py", line 1023 in join
  File "/cpython/portscanner.py", line 26 in main
  File "/cpython/portscanner.py", line 32 in <module>
[1] 63540 segmentation fault ./python -X dev portscanner.py
```

This feature is also helpful when developing and testing C extensions for CPython.

Compiling Debug Support

To get meaningful information from the debugger, the debug symbols must be compiled into CPython. Without these symbols, the stack traces within a debug session won't contain the correct function names, the variable names, or file names.

Windows

Following the same steps as you did in the chapter on Compiling CPython (Windows), ensure that you have compiled in the `Debug` configuration to get the debug symbols:

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

Remember, the `Debug` configuration produces the executable `python_d.exe`, so make sure you use this executable for debugging.

macOS/Linux

The steps in the chapter on Compiling CPython, specify to run the `./configure` script with the `--with-pydebug` flag. If you did not include this flag, go back now and run `./configure` again with your original options and the `--with-pydebug` flag. This will produce the correct executable and symbols for debugging.

Using Lldb for macOS

The `lldb` debugger comes with the Xcode developer tools, so by now you, will have it installed.

Start `lldb` and load the CPython compiled binary as the target:

```
$ lldb ./python.exe
(lldb) target create "./python.exe"
Current executable set to './python.exe' (x86_64).
```

You will now have a prompt where you can enter some commands for debugging.

Creating Breakpoints

To create a breakpoint, use the `break set` command, with the file (relative to the root) and the line number:

```
(lldb) break set --file Objects/floatobject.c --line 532
Breakpoint 1: where = python.exe`float_richcompare + 2276 at
    floatobject.c:532:26, address = 0x000000010006a974
```

Note

There is also a short-hand version of setting breakpoints,
e.g. (lldb) b Objects/floatobject.c:532

You can add multiple breakpoints using the `break set` command. To list the current breakpoints, use the `break list` command:

```
(lldb) break list
Current breakpoints:
1: file = 'Objects/floatobject.c', line = 532, exact_match = 0, locations = 1
  1.1: where = python.exe`float_richcompare + 2276 at floatobject.c:532:26,
        address = python.exe[...], unresolved, hit count = 0
```

Starting CPython

To start CPython, use the `process launch --` command with the command-line options you would normally use for Python, e.g.:

To start python with a string, e.g. `python -c "print(1)"`, use:

```
(lldb) process launch -- -c "print(1)"
```

To start python with a script, use:

```
(lldb) process launch -- my_script.py
```

Attaching to a Running CPython Interpreter

If you have a CPython interpreter running already, you can attach to it.

From inside the lldb session, run `process attach --pid` with the process id:

```
(lldb) process attach --pid 123
```

You can get the process ID from the Activity Monitor, or using `os.getpid()` in Python.

Any breakpoints setup before this point or afterward will halt the process.

An Example of Handling a Breakpoint

To see how breakpoints are handled, set a breakpoint on the `objects` `floatobject.c` `float_richcompare()` function. Next run the process and compare 2 float values using the almost-equal operator that you developed during this book:

```
(lldb) process launch -- -c "1.0~=1.1"
Process 64421 launched: '/cpython/python.exe' (x86_64)
Process 64421 stopped
* thread #1, queue = '...', stop reason = breakpoint 1.1
    frame #0: 0x000000010006a974 python.exe`float_richcompare(v=1.0,
w=1.1, op=6) at floatobject.c:532:26
529         break;
530     case Py_ALE: {
531         double diff = fabs(i - j);
-> 532         const double rel_tol = 1e-9;
533         const double abs_tol = 0.1;
534         r = (((diff <= fabs(rel_tol * j)) ||
```

Target 0: (python.exe) stopped.

lldb will give you a prompt again. You can see the local variables by using the command `v`:

```
(lldb) v
(PyObject *) v = 0x000000010111b370 1.0
(PyObject *) w = 0x000000010111b340 1.1
(int) op = 6
(double) i = 1
(double) j = 1.1000000000000001
```

```
(int) r = 0
(double) diff = 0.10000000000000009
(const double) rel_tol = 2.1256294105914498E-314
(const double) abs_tol = 0
```

You can evaluate a C expression using the `expr` command with any valid C command. The variables in scope can be used. For example, to call `fabs(rel_tol)` and cast to a double, run:

```
(lldb) expr (double)fabs(rel_tol)
(double) $1 = 2.1256294105914498E-314
```

This prints the resulting variable and assigns it an identifier (`$1`). You can reuse this identifier as a temporary variable.

You may also want to explore `PyObject` instances, e.g.:

```
(lldb) expr v->ob_type->tp_name
(const char *) $6 = 0x000000010034fc26 "float"
```

To get a traceback from the breakpoint, use the command `bt`:

```
(lldb) bt
* thread #1, queue = '...', stop reason = breakpoint 1.1
* frame #0: ...
  python.exe`float_richcompare(...) at floatobject.c:532:26
frame #1: ...
  python.exe`do_richcompare(...) at object.c:796:15
frame #2: ...
  python.exe`PyObject_RichCompare(...) at object.c:846:21
frame #3: ...
  python.exe`cmp_outcome(...) at ceval.c:4998:16
```

To step-in, use the command `step`, or `s`.

To continue to the next statement (step-over), use the command `next`, or `n`.

To continue execution, use the command `continue`, or `c`.

To exit the session, use the command `quit`, or `q`.

See Also

The [LLVM Documentation Tutorial](#) contains a more exhaustive list of commands.

Using the Python-Lldb Extension

lldb supports extensions, written in Python. There is an open-source extension which prints additional information in the lldb session for native CPython objects.

To install it, run these commands:

```
$ mkdir -p ~/.lldb
$ cd ~/.lldb && git clone https://github.com/malor/cpython-lldb
$ echo "command script import ~/.lldb/cpython-lldb/cpython_lldb.py"
      >> ~/.lldbinit
$ chmod +x ~/.lldbinit
```

Now, whenever you see variables in lldb, there will be some additional information to the right, such as the numeric value for ints and floats, or the text for Unicode strings. Within a lldb console, there is now an additional command, `py-bt`, that prints the stack trace for Python frames.

Using Gdb

Gdb is a commonly-used debugger for C/C++ applications written on Linux platforms. It is also very popular with the CPython core development team.

When CPython is compiled, it generates a script, `cpython-pdb.py`. Don't execute this script directly. Instead, gdb will discover it and run it automatically once configured. To configure this stage, edit the `.gdbinit` file inside your home path and add the line:

```
add-auto-load-safe-path /path/to/checkout
```

Where `/path/to/checkout` is the path to the `cpython` git checkout.

To start gdb, run it with the argument pointing to your compiled CPython binary.

```
$ gdb ./python
```

Gdb will load the symbols for the compiled binary and give you a command prompt. Gdb has a set of built-in commands, and the CPython extensions bundle some additional commands.

Creating Breakpoints

To set a breakpoint, use the `b <file>:<line>` command, relative to the path of the executable:

```
(gdb) b Objects/floatobject.c:532
Breakpoint 1 at 0x10006a974: file Objects/floatobject.c, line 532.
```

You can set as many breakpoints as you wish.

Starting CPython

To start the process, use the `run` command followed by arguments to start the Python interpreter.

For example, to start with a string:

```
(gdb) run -c "print(1)"
```

To start python with a script, use:

```
(gdb) run my_script.py
```

Attaching to a Running CPython Interpreter

If you have a CPython interpreter running already, you can attach to it.

From inside the gdb session, run `attach` with the process id:

```
(gdb) attach 123
```

You can get the process ID from the Activity Monitor, or using `os.getpid()` in Python.

Any breakpoints setup before this point or afterward will halt the process.

Handling a Breakpoint

When a breakpoint is hit, you can use the `print`, or `p` command to print a variable:

```
(gdb) p *(PyLongObject*)v
$1 = {ob_base = {ob_base = {ob_refcnt = 8, ob_type = ...}, ob_size = 1},
ob_digit = {42}}
```

To step into the next statement, use the command `step`, or `s`. To step over the next statement, use the command `next`, or `n`.

Using the Python-Gdb Extension

The `python-gdb` extension will load an additional command set into the `gdb` console:

Command	Purpose
<code>py-print</code>	Looks up a Python variable and prints it
<code>py-bt</code>	Prints a Python stack trace
<code>py-locals</code>	Prints the result of <code>locals()</code>
<code>py-up</code>	Go down one Python frame

Command	Purpose
py-down	Go up one Python frame
py-list	Print the Python source code for the current frame

Using Visual Studio Debugger

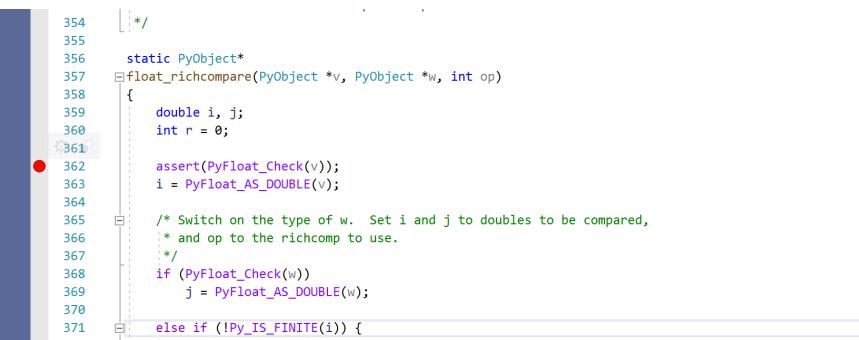
Microsoft Visual Studio comes bundled with a visual debugger. This debugger is powerful, supports a frame stack visualizer, a watch list, and the ability to evaluate expressions.

Open Visual Studio and the `PCBuild\pcbuild.sln` solution file.

Adding Breakpoints

To add a new breakpoint, navigate to the file you want in the solution window, then click in the gutter to the left of the line number.

This adds a red circle to indicate a breakpoint has been set on this line:

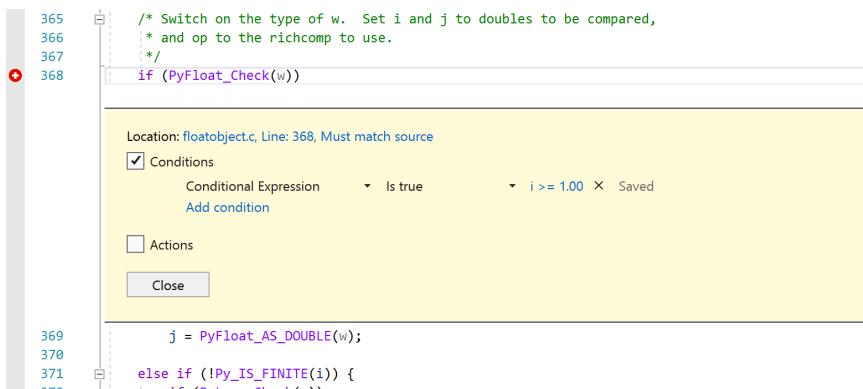


```

354     */
355
356     static PyObject*
357     float_richcompare(PyObject *v, PyObject *w, int op)
358     {
359         double i, j;
360         int r = 0;
361
362         assert(PyFloat_Check(v));
363         i = PyFloat_AS_DOUBLE(v);
364
365         /* Switch on the type of w.  Set i and j to doubles to be compared,
366          * and op to the richcomp to use.
367         */
368         if (PyFloat_Check(w))
369             j = PyFloat_AS_DOUBLE(w);
370
371         else if (!Py_ISFINITE(i)) {
372             r = -1;
373         }
374     }

```

When you hover over the red circle, a cog appears. Click on this cog to configure conditional breakpoints. Add one or more conditional expressions which must evaluate before this breakpoint hits:



```
365  /* Switch on the type of w.  Set i and j to doubles to be compared,
366  * and op to the richcomp to use.
367  */
368  if (PyFloat_Check(w))
```

Location: floatobject.c, Line: 368, Must match source

Conditions

Conditional Expression

Actions

```
369
370
371  j = PyFloat_AS_DOUBLE(w);
372
373  else if (!Py_ISFINITE(i)) {
374      if (PyFloat_Check(w))
```

Starting the Debugger

From the top menu, select **Debug** **Start Debugger**, or press **F5**.

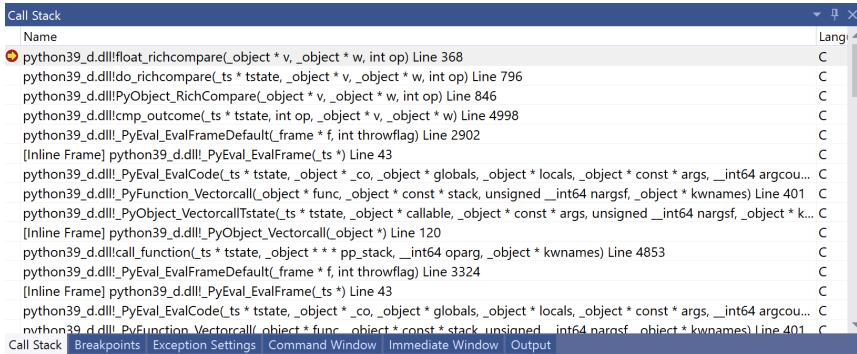
Visual Studio will start a new Python runtime and REPL.

Handling a Breakpoint

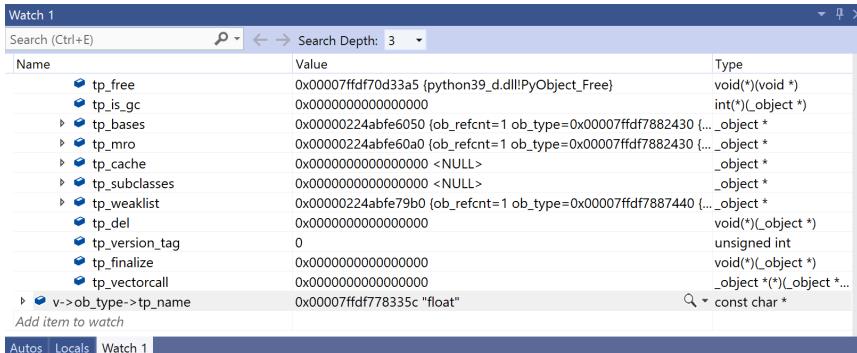
When your breakpoint is hit, you can step forward and into statements using the navigation buttons, or the shortcuts:

- Step Into **F11**
- Step Over **F10**
- Step Out **Shift**+**F11**

At the bottom, a call stack will be shown. You can select frames in the stack to change the navigation and inspect variables in other frames:



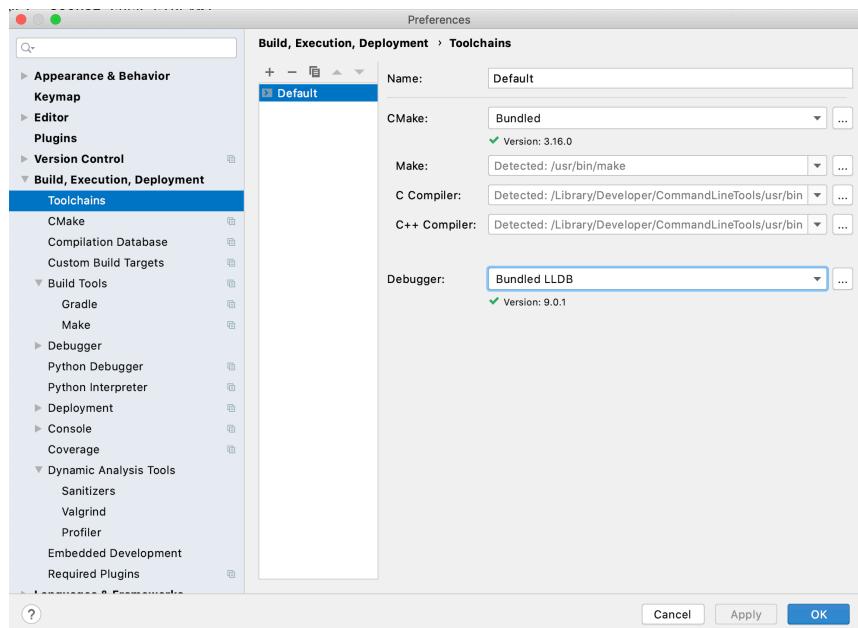
In the code editor, you can highlight any variable or expression to see its value. You can also right-click and choose “Add Watch.” This adds the variable to a list called the Watchlist, where you can quickly see the values of variables you need to help you debug:



Using CLion Debugger

The CLion IDE comes with a powerful visual debugger bundled. It works with lldb on macOS, and gdb on macOS, Windows, and Linux.

To configure the debugger, go to Preferences and select **Build, Execution, Deployment** **Toolchains**:



There is a selection box for the target debugger. Select one of the options:

- For macOS use the “Bundled LLDB”
- For Windows or Linux, use the “Bundled GDB”

Important

Both the LLDB and GDB support benefit from the `cpython-lldb` and `python-gdb` extensions, respectively. Read the LLDB and GDB sections in this chapter for information on how to install and enable these extensions.

Configuring the Custom Build Targets

With your toolchain configured, you can now create Custom Build Targets to enable debugging.

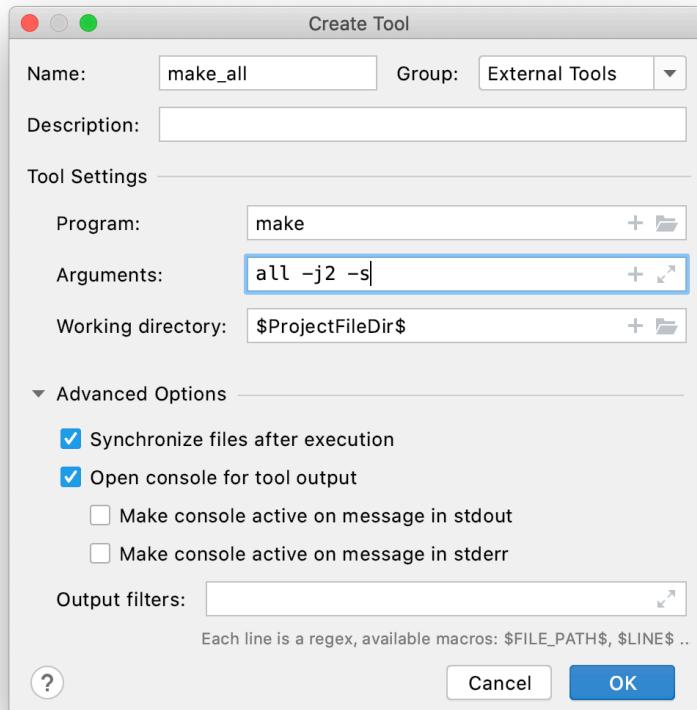
To configure the debugger, go to Preferences and select **Build, Execution, Deployment**  **Toolchains**:

Create a new custom build profile called `cpython_build`, and set the toolchain as `Use Default` to ensure it uses the debugger you just specified.

Select the  next to the Build drop-down to show the **External Tools** window and select  to create a new External Tool.

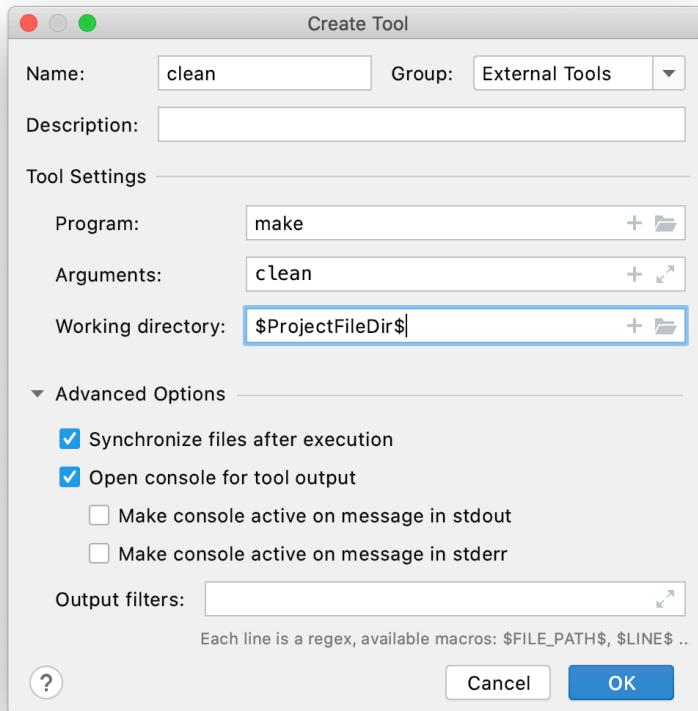
The first external tool will run make

- Set the **Name** as `make_all`
- Set the **Program** as `make`
- Set the **Arguments** as those you have previously used to run make, e.g., `all -j2 -s`
- Set the **Working Directory** as `$ProjectFileDir$`

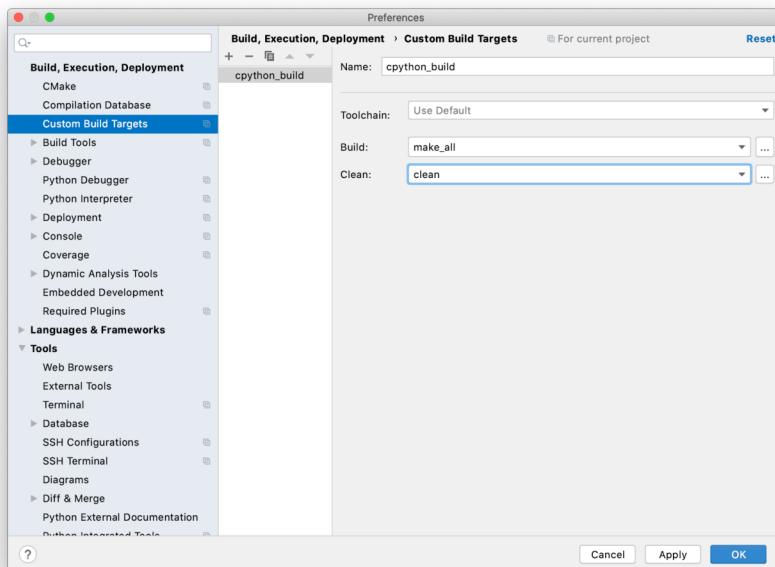


Click **OK** to add the tool, then add a second for cleaning the project:

- Set the **Name** as `clean`
- Set the **Program** as `make`
- Set the **Arguments** as those you have previously used to run `make`, e.g., `clean`
- Set the **Working Directory** as `$ProjectFileDir$`



Close the External tools window and select your `make_all` tool as the **Build** tool and `clean` as the **Clean** tool:



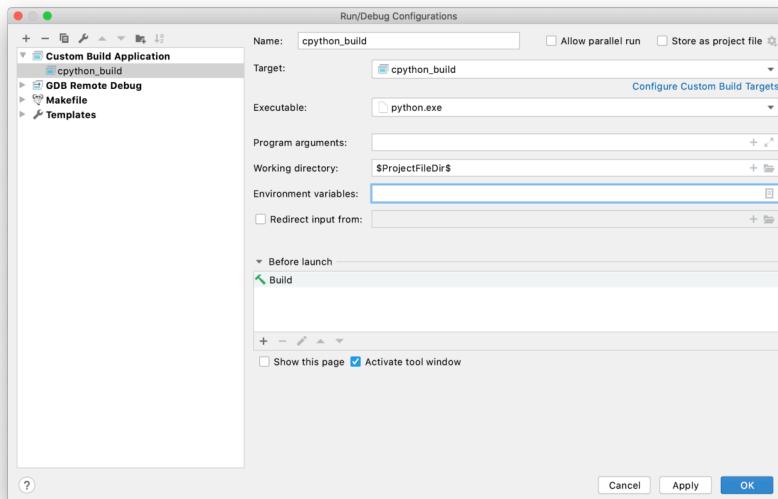
Once this task is completed, you can build, rebuild and clean from the **Build** menu.

Configuring the Custom Debug Target

To debug the compiled CPython executable from the Run/Debug Configurations, open the configuration panel from **Run** \rightarrow **Edit Configurations...**.

Add a new configuration by selecting **[+]** **Custom Build Application**:

- Set the **Target** as `cpython_build`, the Custom Build Target you just created
- Set the **Executable** as `python` for Linux and `python.exe` for macOS
- Set the **Working Directory** as `$ProjectFileDir$`



With this Run/Debug Configuration, you can now debug directly from the **Run** **Debug** menu.

Alternatively, you can attach the debugger to a running CPython process.

Attaching the Debugger

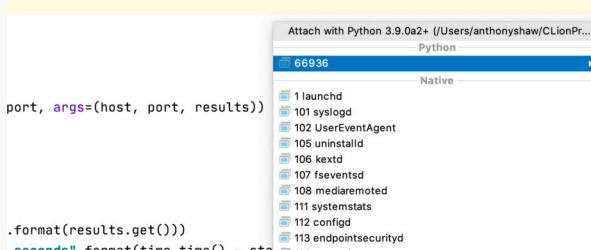
To attach the CLion debugger to a running CPython process, select **Run** **Attach to Process**.

A list of running processes will pop-up. Find the python process you want to attach to and select **Attach**. The debugging session will begin.

Important

If you have the Python plugin installed, it will show the python process at the top. Don't select this one!

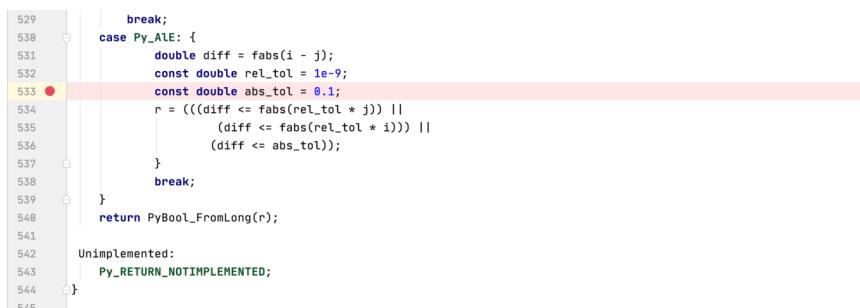
This uses the Python debugger, not the C debugger.



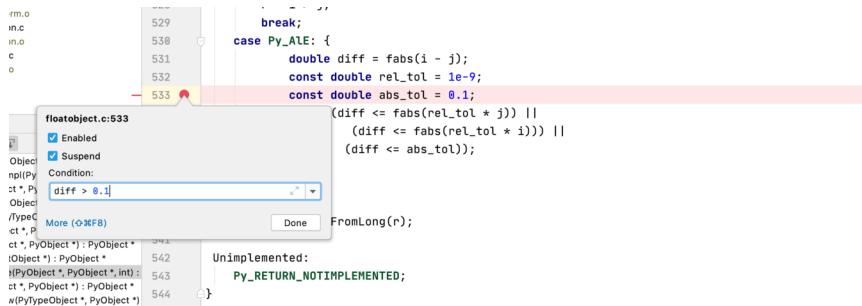
Instead, scroll further down into the “Native” list and find the correct python process.

Creating Breakpoints

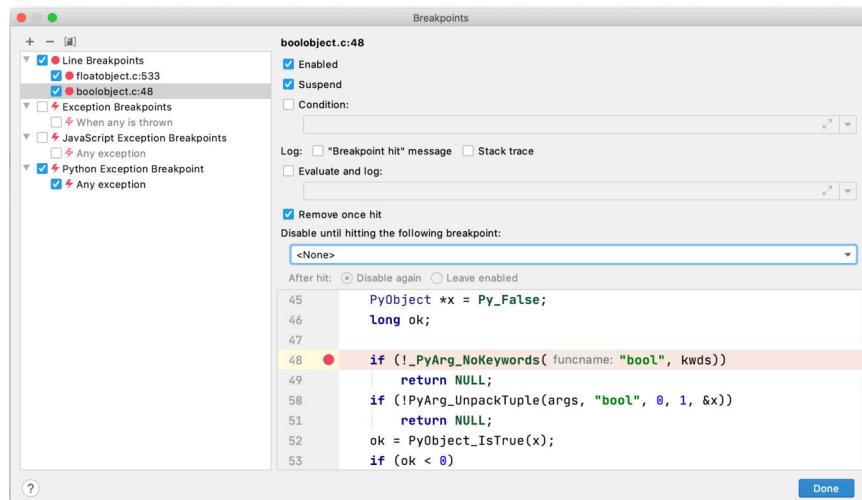
To create a breakpoint, navigate to the file and line you want, then click in the gutter between the line number and the code. A red circle will appear to indicate the breakpoint is set:



Right-click on the breakpoint to attach a condition:



To see and manage all current breakpoints, navigate from the top menu to **Run > View Breakpoints**:

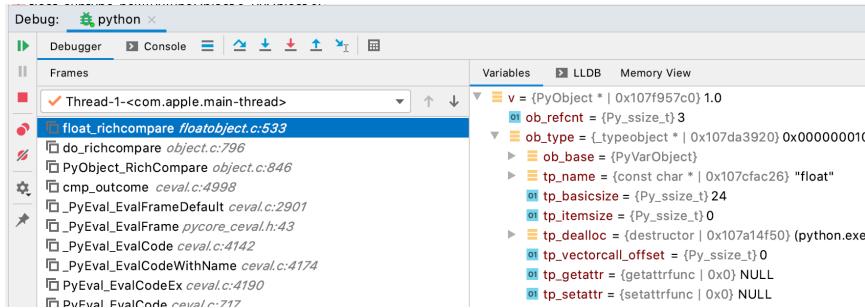


You can enable and disable breakpoints, as well as disable them once another breakpoint has been hit.

Handling Breakpoints

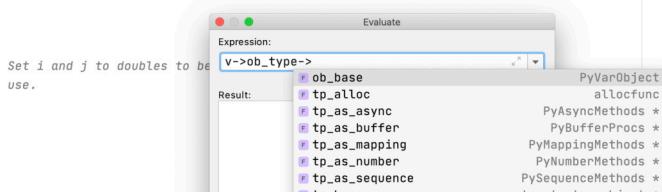
Once a breakpoint has been hit, CLion will set up the Debug panel. Inside the Debug panel is a call stack, showing where the breakpoint hit. You can select other frames in the call stack to switch between them.

Next to the call stack are the local variables. The properties of pointers and type structures can be expanded, and the value of simple types is shown:

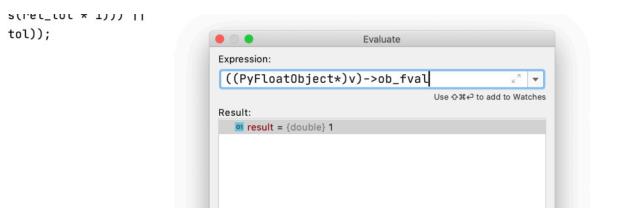


Within a break, you can evaluate expressions to get more information about the local variables. The Evaluation Window can be located in **Run** **>** **Debugging Actions** **>** **Evaluate Expression**, or in a shortcut icon in the Debug Window.

Inside the Evaluate window, you can type expressions and CLion will type-ahead with the property names and types:



You can also cast expressions, which is useful for casting `PyObject*` into the actual type, for example into a `PyFloatObject*`:



Conclusion

In this chapter, you've seen how to set up a debugger on all the major Operating Systems. While the initial setup is time-consuming, the reward is great. Being able to set breakpoints, explore variables, and memory for a running CPython process will give you superpowers. You can use this skill to extend CPython, optimize existing parts of the codebase, or track down nasty bugs.

[Leave feedback on this section »](#)

Benchmarking, Profiling, and Tracing

When making changes to CPython, you need to verify that your changes do not have a significant detrimental impact on performance.

You may want to make changes to CPython that improve performance.

There are solutions for profiling that you will cover in this chapter:

1. Using the `timeit` module to check a simple Python statement thousands of times for the median execution speed
2. Running the Python Benchmark suite to compare multiple versions of Python
3. Using `cProfile` to analyze execution times of frames
4. Profiling the CPython execution with probes

The choice of solution depends on the type of task:

- A **benchmark** will produce an average/median runtime of a fixed code snippet so that you can compare multiple versions of Python runtime
- A **profiler** will produce a call graph, with execution times so that you can understand which function is the slowest

Profilers are available at a C or Python level. If you are profiling a function, module, or script written in Python, you want to use a Python profiler. If you are profiling a C extension module or a modification to the C code in CPython, you need to use a C profiler (or a combination).

Here is a summary of some of the tools available:

Tool	Category	Level	OS Support
timeit	Benchmarking	Python	All
pyperformance	Benchmarking	Python	All
cProfile	Profiling	Python	All
dtrace	Tracing/Profiling	C	Linux/ macOS

Important

Before you run any benchmarks, it is best to close down all applications on your computer so the CPU is dedicated to the benchmark.

Using Timeit for Micro-Benchmarks

The Python Benchmark suite is a thorough test of CPython's runtime with multiple iterations. If you want to run a quick, simple comparison of a specific snippet, use the `timeit` module.

To run `timeit` for a short script, run the compiled CPython with the `-m timeit` module and a script in quotes:

```
$ ./python -m timeit -c "x=1; x+=1; x**x"  
1000000 loops, best of 5: 258 nsec per loop
```

To run a smaller number of loops, use the `-n` flag:

```
$ ./python -m timeit -n 1000 "x=1; x+=1; x**x"  
1000 loops, best of 5: 227 nsec per loop
```

Example

In this book, you have introduced changes to the `float` type by supporting the almost-equal operator.

Try this test to see the current performance of comparing two float values:

```
$ ./python -m timeit -n 1000 "x=1.0001; y=1.0000; x~=y"  
1000 loops, best of 5: 177 nsec per loop
```

The implementation of this comparison is in `float_richcompare()`, inside `Objects/floatobject.c`:

`Objects/floatobject.c` line 358

```
static PyObject*  
float_richcompare(PyObject *v, PyObject *w, int op)  
{  
    ...  
    case Py_Ale: {  
        double diff = fabs(i - j);  
        double rel_tol = 1e-9;  
        double abs_tol = 0.1;  
        r = (((diff <= fabs(rel_tol * j)) ||  
              (diff <= fabs(rel_tol * i))) ||  
              (diff <= abs_tol));  
    }  
    break;  
}
```

Notice that the `rel_tol` and `abs_tol` values are constant, but haven't been marked as constant. Change them to:

```
const double rel_tol = 1e-9;  
const double abs_tol = 0.1;
```

Now, compile CPython again and re-run the test:

```
$ ./python -m timeit -n 1000 "x=1.0001; y=1.0000; x-=y"  
1000 loops, best of 5: 172 nsec per loop
```

You might notice a minor (1-5%) improvement in performance.

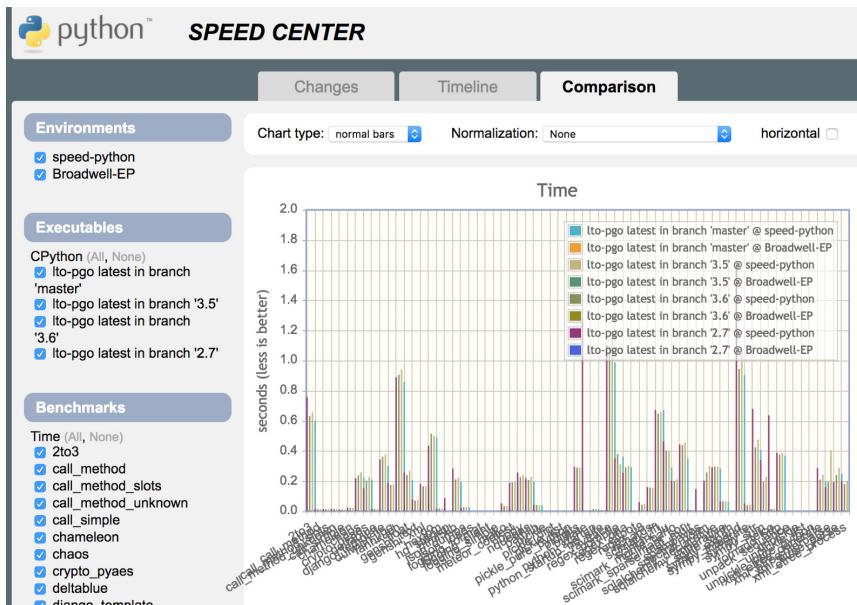
Experiment with different implementations of the comparison to see if you can improve it further.

Using the Python Benchmark Suite for Runtime Benchmarks

The benchmark suite is the tool to use when comparing the complete performance of Python. The Python Benchmark suite is a collection of Python applications designed to test multiple aspects of the Python runtime under load. The Benchmark suite tests are pure-Python, so they can be used to test multiple runtimes, like PyPy and Jython. They are also compatible with Python 2.7 through to the latest version.

Any commits to the master branch on github.com/python/cpython will be tested using the benchmark tool, and the results uploaded to the [Python Speed Center at speed.python.org](https://speed.python.org):

Using the Python Benchmark Suite for Runtime Benchmarks



You can compare commits, branches, and tags side by side on the speed center. The benchmarks use both the Profile Guided Optimization and regular builds with a fixed hardware configuration to produce stable comparisons.

To install the Python benchmark suite, install it from PyPi using a Python runtime (not the one you are testing) in a virtual environment:

```
(venv) $ pip install performance
```

Next, you need to create a configuration file and an output directory for the test profile. It is recommended to create this directory outside of your Git working directory. This also allows you to checkout multiple versions.

In the configuration file, e.g., `~/benchmarks/benchmark.cfg`, put the following contents:

```
cpython-book-samples▶62▶benchmark.cfg
```

```
[config]
# Path to output json files
json_dir = ~/benchmarks/json

# If True, compile CPython in debug mode (LTO and PGO disabled),
# run benchmarks with --debug-single-sample, and disable upload.
#
# Use this option used to quickly test a configuration.
debug = False

[scm]
# Directory of CPython source code (Git repository)
repo_dir = ~/cpython

# Update the Git repository (git fetch)?
update = False

# Name of the Git remote, used to create revision of
# the Git branch.
git_remote = remotes/origin

[compile]
# Create files into bench_dir:
bench_dir = ~/benchmarks/tmp

# Link Time Optimization (LTO)?
lto = True

# Profiled Guided Optimization (PGO)?
pgo = True

# The space-separated list of libraries that are package-only
pkg_only =

# Install Python? If false, run Python from the build directory
install = True

[run_benchmark]
```

```
# Run "sudo python3 -m pyperf system tune" before running benchmarks?
system_tune = True

# --benchmarks option for 'pyperformance run'
benchmarks =

# --affinity option for 'pyperf system tune' and 'pyperformance run'
affinity =

# Upload generated JSON file?
upload = False

# Configuration to upload results to a Codespeed website
[upload]
url =
environment =
executable =
project =

[compile_all]
# List of CPython Git branches
branches = default 3.6 3.5 2.7

# List of revisions to benchmark by compile_all
[compile_all_revisions]
# list of 'sha1=' (default branch: 'master') or 'sha1=branch'
# used by the "pyperformance compile_all" command
```

Executing the Benchmark

To run the benchmark, then run:

```
$ pyperformance compile -U ~/benchmarks/benchmark.cfg HEAD
```

This will compile CPython in the directory you specified and create the JSON output with the benchmark data in the directory specified in the config file.

Comparing Benchmarks

If you want to compare JSON results, the Python Benchmark suite doesn't come with a graphing solution. Instead, you can use this script from within a virtual environment.

To install the dependencies, run:

```
$ pip install seaborn pandas performance
```

Then create a script `profile.py`:

```
cpython-book-samples▶62▶profile.py
```

```
import argparse
from pathlib import Path
from perf._bench import BenchmarkSuite

import seaborn as sns
import pandas as pd

sns.set(style="whitegrid")

parser = argparse.ArgumentParser()
parser.add_argument('files', metavar='N', type=str, nargs='+',
                    help='files to compare')
args = parser.parse_args()

benchmark_names = []
records = []
first = True
for f in args.files:
    benchmark_suite = BenchmarkSuite.load(f)
    if first:
        # Initialise the dictionary keys to the benchmark names
        benchmark_names = benchmark_suite.get_benchmark_names()
        first = False
    bench_name = Path(benchmark_suite.filename).name
    for name in benchmark_names:
```

```
try:
    benchmark = benchmark_suite.get_benchmark(name)
    if benchmark is not None:
        records.append({
            'test': name,
            'runtime': bench_name.replace('.json', ''),
            'stdev': benchmark.stdev(),
            'mean': benchmark.mean(),
            'median': benchmark.median()
        })
except KeyError:
    # Bonus benchmark! ignore.
    pass

df = pd.DataFrame(records)

for test in benchmark_names:
    g = sns.factorplot(
        x="runtime",
        y="mean",
        data=df[df['test'] == test],
        palette="YlGnBu_d",
        size=12,
        aspect=1,
        kind="bar")
    g.despine(left=True)
    g.savefig("png/{}-result.png".format(test))
```

Then, to create a graph, run the script from the interpreter with the JSON files you have created:

```
$ python profile.py ~/benchmarks/json/HEAD.json ...
```

This will produce a series of graphs for each executed benchmark, in the subdirectory `png/`.

Profiling Python Code with cProfile

The standard library comes with two profilers for profiling Python code. The first is `profile`, a pure-Python profiler, and the second, `cProfile`, a faster profiler written in C. In most cases, `cProfile` is the best module to use.

The `cProfile` profiler is used for analyzing a running application and collecting deterministic profiles on the frames evaluated. The output from `cProfile` can be summarized on the command-line, or saved into a `.pstat` file for analysis in an external tool.

In the Parallelism and Concurrency chapter, you wrote a port scanner application in Python. Try profiling that application in `cProfile`.

Run `python` at the command-line with the `-m cProfile` argument to run the `cProfile` module. The second argument is the script to execute:

```
$ python -m cProfile portscanner_multithreaded.py
Port 80 is open
Completed scan in 19.8901150226593 seconds
    6833 function calls (6787 primitive calls) in 19.971 seconds

Ordered by: standard name

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000    0.000    0.000  ...

```

The output will print a table with the columns:

Column	Purpose
<code>ncalls</code>	Number of calls
<code>tottime</code>	Total time spent in the function (minus subfunctions)
<code>percall</code>	Quotient of <code>tottime</code> divided by <code>ncalls</code>
<code>cumtime</code>	Total time spent in this (and subfunctions)
<code>percall</code>	Quotient of <code>cumtime</code> divided by primitive calls
<code>filename:lineno(function)</code>	Data of each function

You can add the `-s` argument and the column name to sort the output. E.g.:

```
$ python -m cProfile -s tottime portscanner_multithreaded.py
```

Will sort by the total time spent in each function.

Exporting Profiles

You can run the `cProfile` module again with the `-o` argument specifying an output file path:

```
$ python -m cProfile -o out.pstat portscanner_multithreaded.py
```

This will create a file, `out.pstat`, that you can load and analyze with the [Stats class](#) or with an external tool.

Visualizing with Snakeviz

`Snakeviz` is a free Python package for visualizing the profile data inside a web browser.

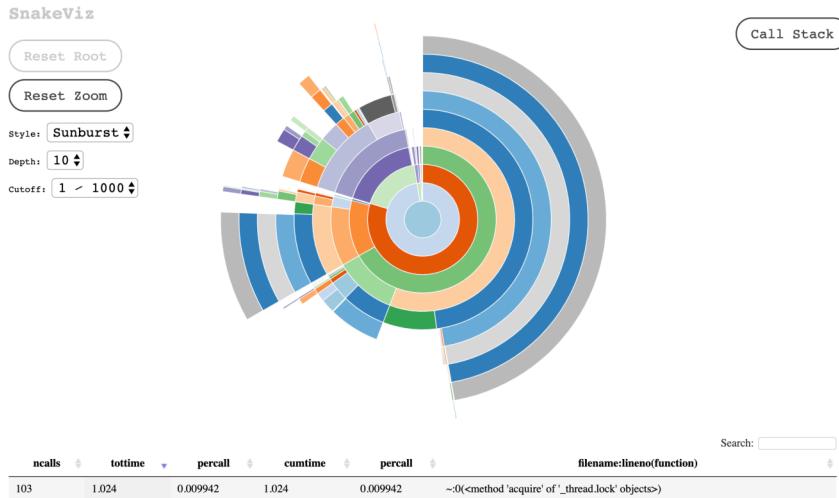
To install `snakeviz`, use `pip`:

```
$ python -m pip install snakeviz
```

Then execute `snakeviz` on the command line with the path to the `stats` file you created:

```
$ python -m snakeviz out.pstat
```

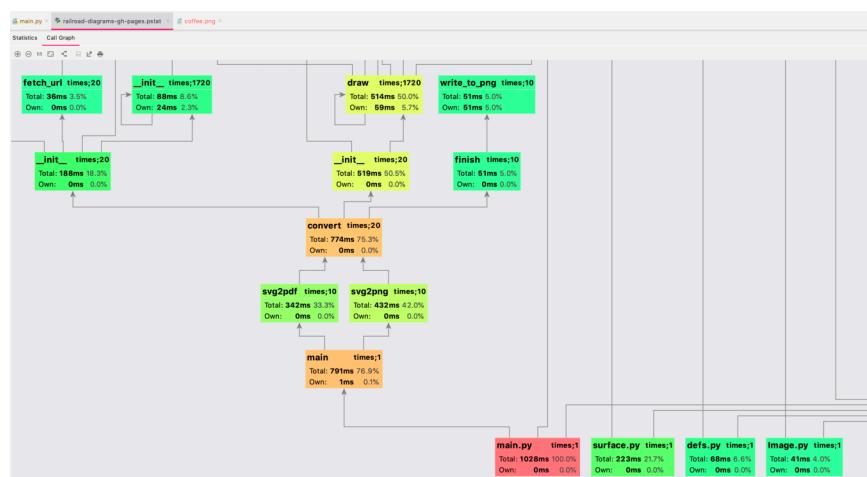
This will open up your browser and allow you to explore and analyze the data:



Visualizing with PyCharm

PyCharm has a builtin tool for running cProfile and visualizing the results. To execute this, you need to have a Python target configured.

To run the profiler, select your run target, then on the top menu select **Run > Profile (target)**. This will execute the run target with cProfile and open a visualization window with the tabular data and a call graph:



Profiling C Code with Dtrace

The CPython source code has several markers for a tracing tool called dtrace. dtrace executes a compiled C/C++ binary, then catches and handles events within it using **probes**.

For dtrace to provide any meaningful data, the compiled application must have custom **markers** compiled into the application. These are events raised during the runtime. The markers can attach arbitrary data to help with tracing.

For example, inside the frame evaluation function, in `Python/ceval.c`, there is a call to `dtrace_function_entry()`:

```
if (PyDTrace_FUNCTION_ENTRY_ENABLED())
    dtrace_function_entry(f);
```

This raises a marker called `function__entry` in dtrace every time a function is entered.

CPython has builtin markers for:

- Line execution
- Function entry and return (frame execution)
- Garbage collection start and completion
- Module import start and completion
- Audit hook events from `sys.audit()`

Each of these markers has arguments with more information. The `function__entry` marker has arguments for:

- File name
- Function name
- Line number

The static marker arguments are defined in the [Official Documentation](#)

dtrace can execute a script file, written in D to execute custom code when probes are triggered. You can also filter out probes based on their attributes.

Related Source Files

Source files related to dtrace are:

File	Purpose
Include <code>pydtrace.h</code>	API definition for dtrace markers
Include <code>pydtrace.d</code>	Metadata the python provider that dtrace uses
Include <code>pydtrace_probes.h</code>	Auto-generated headers for handling probes

Installing Dtrace

dtrace comes pre-installed on macOS, and can be installed in Linux using one of the packaging tools:

For `yum` based systems:

```
$ yum install systemtap-sdt-devel
```

Or, for `apt` based systems:

```
$ apt-get install systemtap-sdt-dev
```

Compiling Dtrace Support

dtrace support must be compiled into CPython. This is done by the `./configuration` script.

Run `./configure` again with the same arguments you used in Compiling CPython and add the flag `--with-dtrace`. Once this is complete, you need to run `make clean && make` to rebuild the binary.

Check that the probe header was created by the configuration tool:

```
$ ls Include/pydtrace_probes.h
Include/pydtrace_probes.h
```

Important

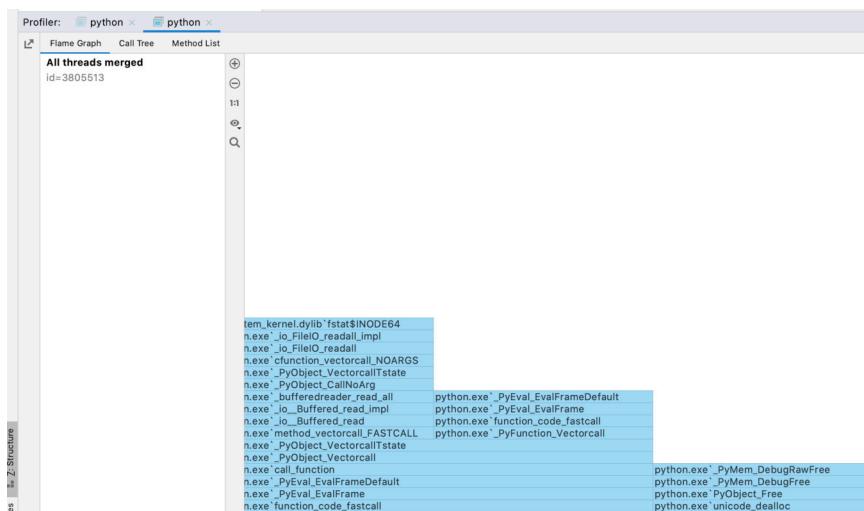
Newer versions of macOS have kernel-level protection that interferes with dtrace called System Integrity Protection.

The examples in this chapter use the CPython probes. If you want to include libc or syscall probes to get extra information, you will need to disable SIP.

Using Dtrace From CLion

The CLion IDE comes bundled with dtrace support. To start tracing, select **Run** **Attach Profiler to Process..** and select the running Python process.

The profiler window will prompt you to start and then stop the tracing session. Once tracing is complete, it provides you with a flame graph showing execution stacks and call times, a Call Tree and a Method List:



Example

In this example, you will test the multithreaded port scanner created in the chapter on Parallelism and Concurrency.

Create a profile script in D, `profile_compare.d`. This profiler will start when `portscanner_multithreaded.py:main()` is entered, to reduce the noise from the interpreter startup.

```
cpython-book-samples▶62▶profile_compare.d
```

```
#pragma D option quiet
self int indent;

python$target:::function-entry
/basename(copyinstr(arg0)) == "portscanner_multithreaded.py"
&& copyinstr(arg1) == "main"/
{
    self->trace = 1;
    self->last = timestamp;
}

python$target:::function-entry
/self->trace/
{
    this->delta = (timestamp - self->last) / 1000;
    printf("%dt%*s:", this->delta, 15, probename);
    printf("%*s", self->indent, "");
    printf("%s:%s:%dn", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
    self->indent++;
    self->last = timestamp;
}

python$target:::function-return
/self->trace/
{
    this->delta = (timestamp - self->last) / 1000;
    self->indent--;
    printf("%dt%*s:", this->delta, 15, probename);
```

```
    printf("%*s", self->indent, "");
    printf("%s:%s:%d", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
    self->last = timestamp;
}

python$target:::function-return
/basename(copyinstr(arg0)) == "portscanner_multithreaded.py"
&& copyinstr(arg1) == "main"/
{
    self->trace = 0;
}
```

This script will print a line every time a function is executed and time the delta between the function starting and exiting.

You need to execute with the script argument, -s profile_compare and the command argument, -c './python portscanner_multithreaded.py':

```
$ sudo dtrace -s profile_compare.d -c './python portscanner_multithreaded.py'
0  function-entry:portscanner_multithreaded.py:main:16
28  function-entry: queue.py:__init__:33
18  function-entry:  queue.py:__init__:205
29  function-return: queue.py:__init__:206
46  function-entry:  threading.py:__init__:223
33  function-return: threading.py:__init__:245
27  function-entry:  threading.py:__init__:223
26  function-return: threading.py:__init__:245
26  function-entry:  threading.py:__init__:223
25  function-return: threading.py:__init__:245
```

In the output, the first column is the time-delta in microseconds since the last event, then the event name, the filename and line number. When function calls are nested, the filename will be increasingly indented to the right.

Conclusion

In this chapter, you have explored benchmarking, profiling, and tracing using a number of tools designed for CPython.

With the right tooling, you can find bottlenecks, compare performance of multiple builds and identify improvements.

[Leave feedback on this section »](#)

Conclusion

Congratulations! You've made it to all the way to the end of this book. There are three main uses of the knowledge you've learned:

1. Writing C extensions
2. Improving your Python applications
3. Contributing to the CPython Project

As a conclusion and summary of the topics, lets explore those.

Writing C Extensions for CPython

There are several ways in which you can extend the functionality of Python. One of these is to write your Python module in C or C++. This process can lead to improved performance and better access to C library functions and system calls.

If you want to write a C extension module, there are some essential things you'll need to know that this book has prepared you for:

- How to set up a C compiler and compile C modules. See the chapter on Compiling CPython.
- How to set up your development environment for C. See the Setting up Your Development Environment chapter.
- How to increment and decrement references to generated objects. See Reference Counting in the Memory Management chapter.

- What `PyObject*` is and its interfaces. See the Object and Variable Object Types section in the Objects and Types chapter.
- What type slots are and how to access Python type APIs from C. See the Type Slots section in the Objects and Types chapter.
- How to add breakpoints to C source files for your extension module and debug them. See the Debugging chapter.

See Also

Over at realpython.com, Danish Prakash has written a great tutorial on [Building a C Extension Module](#). This tutorial includes a concrete example of building, compiling and testing an extension module.

Using This Knowledge to Improve Your Python Applications

There are several important topics covered in this book that can help you improve your applications. Here are some examples:

- Using Parallelism and Concurrency techniques to reduce the execution time of your applications. See the Parallelism and Concurrency chapter.
- Customizing the Garbage Collector algorithm to collect at the end of a task in your application to better handle memory. See the Garbage Collection section in the Memory Management chapter.
- Using the debuggers to debug C extensions and triage issues. See the Debugging chapter.
- Using profilers to profile the execution time of your code. See the Profiling Python Code with `cProfile` section of the Benchmarking, Profiling, and Tracing chapter.
- Analyzing frame execution to inspect and debug complex issues. See the Frame Execution Tracing section in The Evaluation Loop chapter.

Using This Knowledge to Contribute to the CPython Project

While writing this book, CPython had 12 new minor releases, 100's of changes, bug reports, and 1000's of commits to the source code.

CPython is one of the biggest, most vibrant, and open software projects out there. So what sets apart the developers who work on CPython and you, the reader?

Nothing.

That's right; the only thing stopping you from submitting a change, improvement, or fix to CPython is knowing where to start. The CPython community is eager for more contributors. Here are some places you could start:

1. Triaging issues raised by developers on bugs.python.org
2. Fixing small, easy issues

Let's explore each of those in a bit more detail.

Triaging Issues

All bug reports and change requests are first submitted to bugs.python.org. This website is the bug tracker for the CPython Project. Even if you want to submit a pull request on GitHub, you first need a "BPO Number," which is the issue number created by BPO (bugs.python.org).

To get started, register yourself as a user by going to [User](#)  [Register](#) on the left menu.

The default view is not particularly productive and shows both issues raised by users and those raised by core developers (which likely already have a fix).

Instead, after logging in, go to [Your Queries](#)  [Edit](#) on the left menu.

This page will give you a list of queries for the bug index that you can bookmark:

Query	
Patches	leave out
pending issues	leave out
serverhorror's Reports	leave out
Easy Tasks	leave out
Showstoppers	leave out
Latest issues	include
Release Blockers	leave out
Critical	leave out
py3k-open	leave out
Needs review	leave out
Crashers	leave in
Open Doc Bugs 2.6	leave out
Opened patches	leave out
Open documentation issues	leave out

Figure 0.1: bpo-screenshot

Change the value to “leave in” to include these queries in the [Your Queries](#) menu.

Some of the queries I find useful are:

- **Easy Documentation Issues** - showing some documentation improvements that haven’t been made
- **Easy Tasks** - showing some tasks that have been identified as good for beginners
- **Recently Created** - recently created issues
- **Reports without replies** - bug reports that never got a reply
- **Unread** - bug reports that never got read
- **50 latest issues** - the 50 most recently updated issues

With these views, you can follow the [Triaging an Issue](#) guide for the latest process on commenting on issues.

Raising a Pull Request to Fix an Issue

With an issue to fix, you can get started on creating a fix and submitting that fix to the CPython project.

1. Make sure you have a BPO number.
2. Create a branch on your fork of CPython. See the Getting the Source Code chapter for steps on downloading the source code.
3. Create a test to reproduce the issue. See the Testing Modules section of The Test Suite chapter for steps.
4. Make your change following the PEP7 and PEP8 style guides.
5. Run the Regression Test suite to check all the tests are passing. The regression test suite will automatically run on GitHub when you submit the Pull-Request, but it is better to check it locally first. See The Test Suite chapter for steps.
6. Commit and push your changes to GitHub.
7. Go to github.com/python/cpython and create a pull request for your branch

After submitting your pull request it will be triaged by one of the triage teams and assigned to a Core Developer or team for review.

As mentioned earlier, the CPython project needs more contributors. The time between submitting your change can be an hour, a week, or many months. Don't be dismayed if you don't get an immediate response. All of the Core Developers are volunteers and tend to review and merge pull requests in batches.

Important

It is important only to fix one thing in one pull request. If you see a separate (unrelated) issue in some code while writing your patch, make a note and submit it as a second pull request.

To help get your change merged quickly, a good explanation of the problem, the solution, and the fix go a long way.

Other Contributions

Other than bug fixes, there are some different types of improvements you can make to the CPython project:

- Many of the standard library functions and modules are missing unit tests. You can write some tests and submit them to the project.
- Many of the standard library functions don't have up-to-date documentation. You can update the documentation and submit a change.

[Leave feedback on this section »](#)

Keep Learning

Part of what makes Python so great is the community. Know someone learning Python? Help them out! The only way to know you've really mastered a concept is to explain it to someone else.

Come visit us on the web and continue your Python journey on the realpython.com website and the [@realpython](https://twitter.com/realpython) Twitter account.

Weekly Tips for Python Developers

Are you looking for a weekly dose of Python development tips to improve your productivity and streamline your workflows? Good news: we're running a free email newsletter for Python developers just like you.

The newsletter emails we send out are not just your typical list of popular articles. Instead, we aim to share at least one original thought per week in a (short) essay-style format.

If you'd like to see what all the fuss is about, then head on over to realpython.com/newsletter and enter your email address in the signup form. We're looking forward to meeting you!

The Real Python Video Course Library

Become a well-rounded Pythonista with the large (and growing) collection of Python tutorials and in-depth training materials at *Real Python*. With new content published weekly, you'll always find something to boost your skills:

- **Master practical, real-world Python skills:** Our tutorials are created, curated, and vetted by a community of expert Pythonistas. At *Real Python*, you'll get the trusted resources you need on your path to Python mastery.
- **Meet other Pythonistas:** Join the *Real Python* Slack chat and meet the Real Python Team and other subscribers. Discuss your coding and career questions, vote on upcoming tutorial topics, or just hang out with us at this virtual water cooler.
- **Interactive quizzes & Learning Paths:** See where you stand and practice what you learn with interactive quizzes, hands-on coding challenges, and skill-focused learning paths.
- **Track your learning progress:** Mark lessons as completed or in-progress and learn at your own pace. Bookmark interesting lessons and review them later to boost long-term retention.
- **Completion certificates:** For each course you complete, you receive a shareable (and printable) Certificate of Completion, hosted privately on the *Real Python* website. Embed your certificates in your portfolio, LinkedIn resume, and other websites to show the world that you're a dedicated Pythonista.
- **Regularly updated:** Keep your skills fresh and keep up with technology. We're constantly releasing new members-only tutorials and update our content regularly.

See what's available at realpython.com/courses

Appendix 1 : Introduction to C for Python Programmers

This introduction is intended to get an experienced Python programmer up to speed with the basics of the C language and how it's used in the CPython source code. It assumes you've already got an intermediate understanding of Python syntax.

That said, C is a fairly limited language and most of it's usage in CPython falls into a small set of syntax. Getting to the point where you understand the code is a much smaller step than being able to write C effectively. This tutorial is aimed at the first goal but not the second.

One of the first things that stand out as a big difference between Python and C is the C preprocessor. Let's look at that first.

C Preprocessor

The preprocessor, as the name suggests, is run on your source files before the compiler runs. It has very limited abilities, but these can be used to great advantage in building C programs. The preprocessor produces a new file which is what the compiler will actually process. All of the commands to the preprocess start at the beginning of a line with a `#` symbol as the first non-whitespace character.

The main purpose of the preprocessor is to do text substitution in the source file, but it will also do some basic conditional code if `#if` state-

ments.

Let's start with the most frequent preprocessor directive, `#include`.

#include

`#include` is used to pull the contents of one file into the current source file. There is nothing sophisticated about this, it reads that file from the file system, runs the preprocessor on that file and puts the results of that into the output file. This is done recursively for each `#include` directive found.

For example, if you look at the `Modules/_multiprocessing/semaphore.c` file, near the top you'll see:

```
#include "multiprocessing.h"
```

This tells the preprocessor to pull in the entire contents of `multiprocessing.h` and put it into the output file at this position.

You will notice two different forms for the `include` statement. One of them uses quotes to specify the name of the include file, the other uses angle brackets (`<>`). The difference comes from which paths are searched when looking for the file on the file system. If you use `<>` for the filename, the preprocessor will only look on “system” include files. Using quotes around the filename instead will force the preprocessor to look in the local directory first and then fall back to the system directories.

#define

`#define` allows you to do simple text substitution and also plays into the `#if` directives we'll see below.

At its most basic, `#define` lets you define a new symbol that gets replaced with a text string in the preprocessor output.

Continuing in `semaphore.c` you'll find this line:

```
#define SEM_FAILED NULL
```

This tells the preprocessor to replace every instance of `SEM_FAILED` below this point with the literal string `NULL` before the code is sent to the compiler.

`#define` items can also take parameters as in this Windows-specific version of `SEM_CREATE`:

```
#define SEM_CREATE(name, val, max) CreateSemaphore(NULL, val, max, NULL)
```

In this case the preprocessor will expect `SEM_CREATE()` to look like a function call and have three parameters. This is generally referred to as a **macro**. It will directly replace the text of the three parameters into the output code. For example, on line 459, the `SEM_CREATE` macro is used like this:

```
handle = SEM_CREATE(name, value, maxvalue);
```

When compiling for Windows, this macro will be expanded so that line is:

```
handle = CreateSemaphore(NULL, value, max, NULL);
```

We'll see below how this macro is defined differently on Windows and other operating systems.

#undef

This directive erases any previous preprocessor definition from `#define`. This makes it possible to have a `#define` in effect for only part of a file.

#if

The preprocessor also allows conditional statements, allowing you to either include or exclude sections of text based on certain conditions. Conditional statements are closed with the `#endif` directive and also can make use of `#elif` and `#else` for fine-tuned adjustments.

There are three basic forms of `#if` that you'll see in the CPython source:

- `#ifdef <macro>` : includes the following block of text if the specified macro is defined
- `#if defined(<macro>)`: same as `#ifdef`
- `#ifndef <macro>`: includes the following block of text if the specified macro is not defined
- `#if <macro>`: includes the following text if the macro is defined **and** it evaluates to True

Note the use of “text” instead of “code” to describe what is included or excluded from the file. The preprocessor knows nothing of C syntax and does not care what the specified text is.

#pragma

Pragmas are instructions or hints to the compiler. In general you can ignore these while reading the code as they usually deal with how the code is compiled, not how the code runs.

#error

Finally, `#error` displays a message and causes the preprocessor to stop executing. Again, you can safely ignore these for reading the CPython source code.

Basic C Syntax

This section will not cover ALL aspects of C nor, again, is it intended to teach you how to write C. It will focus on things that are different or confusing for Python developers the first time they see them.

General

Unlike in Python, whitespace is not important to the C compiler. It does not care if you split statements across lines or jam your entire program onto a single, very long line. This is because it uses delimiters for all statements and blocks.

There are, of course, very specific rules for the parser, but in general you'll understand the CPython source just knowing that each statement ends with a semicolon (;) and all blocks of code are surrounded by curly braces ({}).

The exception to this rule is that if a block has only a single statement, the curly braces can be omitted.

All variables in C must be **declared** meaning there needs to be a single statement giving the **type** of that variable. Note that, unlike Python, the data type that a single variable can hold cannot change.

Let's look at some examples:

```
/* comments are included between slash-asterisk and asterisk-slash */
/* This style of comment can span several lines -
   so this part is still a comment. */

// OR comments can be after two slashes
// This type of comments only go until the end of the line, so new
// lines must have //s again.

int x = 0; // declares x to be of type 'int' and initializes it to 0

if (x == 0) {
    // this is a block of code
    int y = 1; // y is only a valid variable name until the closing }
    // more statements here
    printf("x is %d y is %dn", x, y);
}

// single line blocks do not require curly brackets
```

```
if (x == 13)
    printf("x is 13!\n");
printf("past the if blockn");
```

In general you'll see that the CPython code is very cleanly formatted and generally sticks to a single style within a given module.

if Statements

In C, `if` works generally like it does in Python. If the condition is true then the following block is executing. The `else` and `else if` syntax should be familiar enough to Python programmers to understand. Note that C `if` statements do not need and `endif` because blocks are delimited by `{}`.

There is a shorthand in C for short if/else statements call the **ternary operator**:

You can find it in `semaphore.c` where, for Windows, it defines a macro for `SEM_CLOSE()`:

```
#define SEM_CLOSE(sem) (CloseHandle(sem) ? 0 : -1)
```

The return value of this macro will be `0` if the function `CloseHandle()` returns true and `-1` otherwise.

Just a note about `true` in C. Boolean variable types are supported and used in parts of the CPython source, but they were not a part of the original language. C interprets binary conditions using a simple rule: `0` or `NULL` is false, everything else is true.

switch Statements

Unlike Python, C also supports `switch`. Using `switch` can be viewed as a shortcut for extended `if/elseif` chains. This example is from `semaphore.c`:

```

switch (WaitForSingleObjectEx(handle, 0, FALSE)) {
    case WAIT_OBJECT_0:
        if (!ReleaseSemaphore(handle, 1, &previous))
            return MP_STANDARD_ERROR;
        *value = previous + 1;
        return 0;
    case WAIT_TIMEOUT:
        *value = 0;
        return 0;
    default:
        return MP_STANDARD_ERROR;
}

```

This performs a switch on the return value from `WaitForSingleObjectEx()`. If the value is `WAIT_OBJECT_0`, the first block is executed. The `WAIT_TIMEOUT` value results in the second block, and anything else matches the `default` block.

Note that the value being tested, in this case the return value from `WaitForSingleObjectEx()`, must be an integral value or an enumerated type and that each `case` must be a constant value.

Loops

There are three looping structures in C:

- `for` loops
- `while` loops
- `do..while` loops

Let's look at each of these in turn.

`for` loops have syntax that is quite different than Python:

```

for ( <initialization>; <condition>; <increment> ) {
    <code to be looped over>
}

```

In addition to the code to be executed in the loop, there are three blocks of code which control the `for` loop.

The `<initialization>` section is run exactly one time when the loop is started. It traditionally is used to set a loop counter to an initial value (and possibly declare the loop counter). The `<increment>` code is run immediately after each pass through the main block of the loop. Traditionally this will increment the loop counter. Finally, the `<condition>` is run after the `<increment>`. The return value of this code will be evaluated and the loop breaks when this condition returns false.

Here's an example from `Modules/sha512module.c`:

```
for (i = 0; i < 8; ++i) {
    S[i] = sha_info->digest[i];
}
```

This loop will run 8 times, with `i` going from 0 to 7, terminating when the condition is checked and `i` is 8.

`while` loops are virtually identical to their Python counterparts. The `do..while()` syntax is a little different, however. The condition on a `do..while()` loop is not checked until after the first time the body of the loop is executed.

There are many instances of `for` loops and `while` loops in the CPython code base, but `do..while()` is unused.

Functions

The syntax for functions in C is similar to that in Python, with the addition that the return type and parameter types must be specified. The C syntax looks like this:

```
<return_type> function_name(<parameters>) {
    <function_body>
}
```

The return type can be any valid type in C, including both built-in

types like `int` and `double` as well as custom types like `PyObject` like in this example from `semaphore.c`:

```
static PyObject *
semlock_release(SemLockObject *self, PyObject *args)
{
    <statements of function body here>
}
```

Here you see a couple of C-specific things in play. First off, remember that whitespace does not matter. Much of the CPython source code puts the return type of a function on the line above the rest of that function declaration. That's the `PyObject *` part. We'll talk about the `*` aspect of this a little later, but here we should point out that there are several modifiers you can place on functions and variables.

`static` is one of these modifiers. Unfortunately these modifiers have some complex rules governing how they operate. For instance, the `static` modifier here means something very different than placing it in front of a variable declarations.

Fortunately, these modifiers can generally be ignored while trying to read and understand the CPython source code.

The parameter list for functions is a comma-separated list of variables, similar to Python. Again, C requires specific types for each parameter, so the `SemLockObject *self` says that the first parameter (all parameters in C are positional) is a pointer to a `SemLockObject` and is called `self`.

Let's look at what the pointer part of that statement means.

To give some context, the parameters that are passed to C are all passed by value, meaning the function operates on a copy of the value and not the original in the calling function. To work around this, functions will frequently pass in the address of some data that the function can modify. These addresses are called **pointers** and have types, so `int *` is a pointer to an `int` value and is of a different type than `double *` which is a pointer to a `double`.

Pointers

As mentioned above, pointers are variables that hold the address of a value. These are used frequently in C as seen in the example above:

```
static PyObject *
semlock_release(SemLockObject *self, PyObject *args)
{
    <statements of function body here>
}
```

Here the `self` parameter will hold the address of (usually called “a pointer to”) a `SemLockObject` value. Also note that the function will return a pointer to a `PyObject` value.

There is a special value in C to indicate that a pointer does not point to anything, called `NULL`. You’ll see pointers assigned to `NULL` and checked against `NULL` throughout the CPython source. This is important as there are very few limitations as to what values a pointer can have and access a memory location that is not officially part of your program can cause very strange behavior.

If you try to access the memory at `NULL`, on the other hand, your program will exit immediately. This may not seem better, but it’s generally easier to figure out a memory bug if `NULL` is accessed than if a random memory address is modified.

Strings

In C there is not a string type. There is a convention around which many standard library functions are written, but there is not an actual type. Rather, strings in C are stored as arrays of `char` or `wchar` values, each of which holds a single character. Strings are marked with a **null-terminator** which has a value `0` and is usually shown in code as `0`.

Basic string operations like `strlen()` rely on this null-terminator to mark the end of the string.

Because strings are just arrays of values, they cannot be directly copied or compared. The standard library has the `strcpy()` and `strcmp()` functions (and their `wchar` cousins) for doing these operations and more.

Structs

Your final stop on this mini-tour of C is how you can create new types in C: **structs**. The `struct` keyword allows you to group a set of different data types together into a new, custom data type:

```
struct <struct_name> {
    <type> <member_name>;
    <type> <member_name>;
    ...
};
```

This partial example from `Modules/arraymodule.c` shows a struct declaration:

```
struct arraydescr {
    char typecode;
    int itemsize;
    ...
};
```

This creates a new data type called `struct arraydescr` which has many members, the first two of which are a `char typecode` and an `int itemsize`.

Frequently structs will be used as part of a `typedef` which provides a simple alias for the name. In the example above, all variables of the new type must be declared with the full name `struct arraydescr x;`.

You'll frequently see syntax like this:

```
typedef struct {
    PyObject_HEAD
    SEM_HANDLE handle;
```

```
unsigned long last_tid;
int count;
int maxvalue;
int kind;
char *name;
} SemLockObject;
```

This creates a new, custom struct type and gives it a name `SemLockObject`. To declare a variable of this type, you can simply use the alias `SemLockObject x;`.

Conclusion

This wraps up the quick walk through C syntax. There were many corners that were cut in this description, but it should be sufficient to read the CPython source code.

[Leave feedback on this section »](#)

This is an Early Access version of “CPython Internals: Your Guide to the Python 3 Interpreter”

With your help we can make this book even better:

At the end of each section of the book you’ll find a “magical” feedback link. Clicking the link takes you to an **online feedback form** where you can share your thoughts with us.

Please feel free to be as terse or detailed as you see fit. All feedback is stored anonymously, but you can choose to leave your name and contact information so we can follow up or mention you on our “Thank You” page.

We use a different feedback link for each section, so we’ll always know which part of the book your notes refer to.

Thank you for helping us make this book an even more valuable learning resource for the Python community.

— Anthony Shaw

Inside The Python Virtual Machine



Obi Ike-Nwosu

Inside The Python Virtual Machine

Obi Ike-Nwosu

This book is for sale at <http://leanpub.com/insidethepythonvirtualmachine>

This version was published on 2020-08-07



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2020 Obi Ike-Nwosu

Also By Obi Ike-Nwosu

Intermediate Python

Contents

1. Introduction	1
2. The View From 30,000ft	3
3. Compiling Python Source Code	8
3.1 From Source To Parse Tree	8
3.2 Python tokens	9
3.3 From Parse Tree To Abstract Syntax Tree	13
3.4 Building The Symbol Table	17
3.5 From AST To Code Objects	24
4. Python Objects	37
4.1 PyObject	37
4.2 Dissecting Types	38
4.3 Type Object Case Studies	41
4.4 Minting type instances	44
4.5 Objects and their attributes	48
4.6 Method Resolution Order (MRO)	59
5. Code Objects	62
5.1 Exploring code objects	62
5.2 Code Objects within other code objects	68
5.3 Code Objects in the VM	70
6. Frame Objects	72
6.1 Allocating Frame Objects	74
7. Interpreter and Thread States	76
7.1 The Interpreter state	76
7.2 The Thread state	78
8. Intermezzo: The <code>abstract.c</code> Module	84
9. The evaluation loop, <code>ceval.c</code>	88
9.1 Putting names in place	88
9.2 The parts of the machine	90

CONTENTS

9.3	The Evaluation loop	94
9.4	A sampling of opcodes	99
10.	The Block Stack	103
10.1	A Short Note on Exception Handling	106
11.	From Class code to bytecode	109
12.	Generators: Behind the scenes.	116
12.1	The Generator object	116
12.2	Running a generator	119

1. Introduction

The Python Programming language has been around for a long time. Guido van Rossum started development work on the first version in 1989, and it has since grown to become one of the more popular languages used in a wide range of applications from graphical interfaces to [finance](#)¹ and [data analysis](#)².

This write-up looks at the nuts and bolts of the Python interpreter. It targets CPython, the most popular, and reference implementation of Python at the point of this write-up.



Python and CPython are used interchangeably in this text, but any mention of Python refers to CPython, the version of Python implemented in C. Other implementations include PyPy - Python implemented in a restricted subset of Python, Jython - Python implemented on the Java Virtual Machine, etc.

I regard the execution of a Python program as split into two or three main phases, as listed below. The relevant stages depend on how the interpreter is invoked, and this write-up covers them in different measures:

1. Initialization: This step covers the set up of the various data structures needed by the Python process and is only relevant when a program is executed non-interactively through the command prompt.
2. Compiling: This involves activities such as building syntax trees from source code, creating the abstract syntax tree, building the symbol tables, generating code objects etc.
3. Interpreting: This involves the execution of the generated code object's bytecode within some context.

The methods used in generating parse trees and syntax trees from source code are language-agnostic, so we do not spend much time on these. On the other hand, building symbol tables and code objects from the *Abstract Syntax tree* is the more exciting part of the compilation phase. This step is more Python-centric, and we pay particular attention to it. Topics we will cover include generating symbol tables, Python objects, frame objects, code objects, function objects etc. We will also look at how code objects are interpreted and the data structures that support this process.

This material is for anyone interested in gaining insight into how the CPython interpreter functions. The assumption is that the reader is already familiar with Python and understands the fundamentals of the language. As part of this exposition, we go through a considerable amount of C code, so a

¹<http://tpq.io/>

²<http://pandas.pydata.org/>

reader with a rudimentary understanding of C will find it easier to follow. All that is needed to get through this material is a healthy desire to learn about the CPython virtual machine.

This work is an expanded version of personal notes taken while investigating the inner working of the Python interpreter. There is a substantial amount of wisdom in videos available in [Pycon videos](#)³, [school lectures](#)⁴ and [blog write-ups](#)⁵. This work will be incomplete without acknowledging these fantastic sources.

At the end of this write-up, a reader should understand the processes and data structures that are crucial to the execution of a Python program. We start next with an overview of the execution of a script passed as a command-line argument to the interpreter. Readers can install the CPython executable from the source by following the instructions at the [Python Developer's Guide](#)⁶.



This material makes use of Python 3.6.

³<https://www.youtube.com/watch?v=XGF3Qu4dUqk>

⁴[http://pgbovine.net/cpython-internals.htm/](http://pgbovine.net/cpython-internals.htm)

⁵<https://tech.blog.aknin.name/2010/04/02/pythons-innards-introduction/>

⁶<https://docs.python.org/devguide/index.html#>

2. The View From 30,000ft

This chapter is a high-level overview of the processes involved in executing a Python program. Regardless of the complexity of a Python program, the techniques described here are the same. In subsequent chapters, we zoom in to give details on the various pieces of the puzzle. The excellent explanation of this process provided by Yaniv Aknin in his [Python Internal series¹](#) provides some of the basis and motivation for this discussion.

A method of executing a Python script is to pass it as an argument to the Python interpreter as such `$python test.py`. There are other ways of interacting with the interpreter - we could start the interactive interpreter, execute a string as code, etc. However, these methods are not of interest to us. Figure 2.1 is the flow of activities involved in executing a module passed to the interpreter at the command-line.

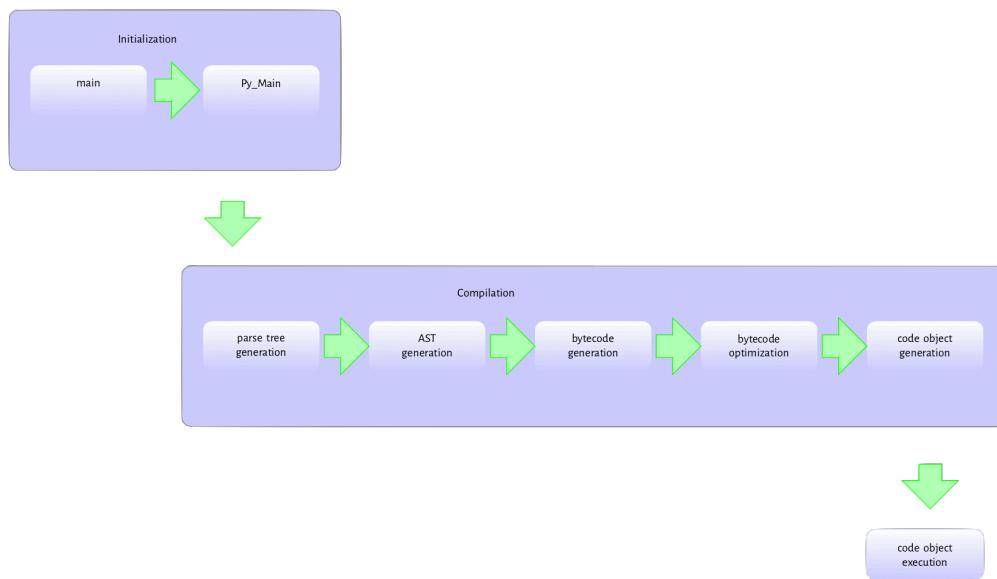


Figure 2.1: Flow during the execution of source code

The Python executable is a C program like any other C program such as the Linux kernel or a simple `hello world` program in C so pretty much the same process happens when we run the Python interpreter executable. The executable's entry point is the `main` method in the `Programs/python.c`. This `main` method handles basic initialization, such as memory allocation, locale setting, etc. Then, it invokes the `Py_Main` function in `Modules/main.c` responsible for the python specific initializations. These include parsing command-line arguments and setting program `flags`², reading environment variables, running hooks, carrying out hash randomization, etc. After, `Py_Main` calls the `Py_Initialize` function in `Programs/pylifecycle.c`; `Py_Initialize` is responsible for initializing

¹<https://tech.blog.aknin.name/2010/04/02/pythons-innards-introduction/>

²<https://docs.python.org/3.6/using/cmdline.html>

the interpreter and all associated objects and data structures required by the Python runtime. After `Py_Initialize` completes successfully, we now have access to all Python objects.



This writeup assumes a Unix based operating system, so some specifics may differ when using a Windows operating system.

The interpreter state and interpreter state data structures are two examples of data structures that are initialized by the `Py_Initialize` call. A look at the data structure definitions for these provides some context into their functions. The interpreter and thread states are C structures with pointers to fields that hold information needed for executing a program. Listing 2.1 is the interpreter state `typedef` (just assume that `typedef` is C jargon for a type definition though this is not entirely true).

Listing 2.1: The interpreter state data structure

```

1  typedef struct _is {
2
3      struct _is *next;
4      struct _ts *tstate_head;
5
6      PyObject *modules;
7      PyObject *modules_by_index;
8      PyObject *sysdict;
9      PyObject *builtins;
10     PyObject *importlib;
11
12     PyObject *codec_search_path;
13     PyObject *codec_search_cache;
14     PyObject *codec_error_registry;
15     int codecs_initialized;
16     int fscodec_initialized;
17
18     PyObject *builtins_copy;
19 } PyInterpreterState;

```

Anyone who has used the Python programming language long enough may recognize a few of the fields mentioned in this structure (`sysdict`, `builtins`, `codec`)^{*}.

1. The `*next` field is a reference to another interpreter instance as multiple python interpreters can exist within the same process.
2. The `*tstate_head` field points to the main thread of execution - if the Python program is multithreaded, then the interpreter is shared by all threads created by the program - we discuss the structure of a thread state shortly.

3. The `modules`, `modules_by_index`, `sysdict`, `builtins`, and `importlib` are self-explanatory - they are all defined as instances of `PyObject` which is the root type of all Python objects in the virtual machine world. We provide more details about Python objects in the chapters that will follow.
4. The `codec*` related fields hold information that helps with the location and loading of encodings. These are very important for decoding bytes.

A Python program must execute in a thread. The thread state structure contains all the information needed by a thread to run some code. Listing 2.2 is a fragment of the thread data structure.

Listing 2.2: A cross-section of the thread state data structure

```
1  typedef struct _ts {  
2      struct _ts *prev;  
3      struct _ts *next;  
4      PyInterpreterState *interp;  
5  
6      struct _frame *frame;  
7      int recursion_depth;  
8      char overflowed;  
9  
10     char recursion_critical;  
11     int tracing;  
12     int use_tracing;  
13  
14     Py_tracefunc c_profilefunc;  
15     Py_tracefunc c_tracefunc;  
16     PyObject *c_profileobj;  
17     PyObject *c_traceobj;  
18  
19     PyObject *curexc_type;  
20     PyObject *curexc_value;  
21     PyObject *curexc_traceback;  
22  
23     PyObject *exc_type;  
24     PyObject *exc_value;  
25     PyObject *exc_traceback;  
26  
27     PyObject *dict; /* Stores per-thread state */  
28     int gilstate_counter;  
29  
30     ...  
31 } PyThreadState;
```

More details on the interpreter and the thread state data structures will follow in subsequent chapters. The initialization process also sets up the import mechanisms as well as rudimentary stdio.

After the initialization, the `Py_Main` function invokes the `run_file` function also in the `main.c` module. The following series of function calls: `PyRun_AnyFileExFlags` -> `PyRun_SimpleFileExFlags`->`PyRun_FileExFlags`->`PyParser_ASTFromObject` are made to the `PyParser_ASTFromObject` function. The `PyRun_SimpleFileExFlags` function call creates the `__main__` namespace in which the file contents will be executed. It also checks for the presence of a `pyc` version of the module - the `pyc` file contains the compiled version of the executing module. If a `pyc` version exists, it will attempt to read and execute it. Otherwise, the interpreter invokes the `PyRun_FileExFlags` function followed by a call to the `PyParser_ASTFromObject` function and then the `PyParser_ParseFileObject` function. The `PyParser_ParseFileObject` function reads the module content and builds a parse tree from it. The `PyParser_ASTFromNodeObject` function is then called with the parse tree as an argument and creates an abstract syntax tree (AST) from the parse tree.



If you have been following the actual C source code, you must have run into the `Py_INCREF` and `Py_DECREF` by now. These are memory management functions that will be discussed later on in more detail. CPython manages the object life cycle using reference counting; whenever we create a new reference to an object, a call to the `Py_INCREF` function is used to increase with the reference count of that object by 1. Whenever a reference goes out of scope, a call to the `Py_DECREF` functions reduces the reference count by 1.

The AST generated is then passed to the `run_mod` function. This function invokes the `PyAST_CompileObject` function that creates code objects from the AST. Do note that the bytecode generated during the call to `PyAST_CompileObject` is passed through a simple peephole optimizer that carries out low hanging optimization of the generated bytecode before creating the code object. With the code objects created, it is time to execute the instructions encapsulated by the code objects. The `run_mod` function invokes `PyEval_EvalCode` from the `ceval.c` file with the code object as an argument. This results in another series of function calls: `PyEval_EvalCode`->`PyEval_EvalCode`->`_PyEval_EvalCodeWithName`->`_PyEval_EvalFrameEx`. The code object is an argument to most of these functions. The `_PyEval_EvalFrameEx` is the actual execution loop that handles executing the code objects. This function gets called with a *frame object* as an argument. This *frame object* provides the context for executing the code object. The execution loop reads and executes instructions from an array of instructions, adding or removing objects from the value stack in the process (*where is this value stack?*), till there are no more instructions to execute or something exceptional that breaks this loop occurs.

Python provides a set of functions that one can use to explore actual code objects. For example, a simple program can be compiled into a code object and disassembled to get the opcodes that are executed by the Python virtual machine, as shown in listing 2.3.

Listing 2.3: Disassembling a python function

```

1      >>> from dis import dis
2      >>> def square(x):
3          ...      return x*x
4          ...
5
6      >>> dis(square)
7          0  LOAD_FAST          0  (x)
8          2  LOAD_FAST          0  (x)
9          4  BINARY_MULTIPLY
10         6  RETURN_VALUE

```

The `./Include/pcodes.h` file contains a listing of the Python Virtual Machine's bytecode instructions. The opcodes are pretty straight forward conceptually. Take our example from listing 2.3 with four instructions - the `LOAD_FAST` opcode loads the value of its argument (`x` in this case) onto an evaluation (value) stack. The Python virtual machine is a stack-based virtual machine, so values for operations and results from operations live on a stack. The `BINARY_MULTIPLY` opcode then pops two items from the value stack, performs binary multiplication on both values, and places the result back on the value stack. The `RETURN_VALUE` opcode pops a value from the stack, sets the return value object to this value, and breaks out of the interpreter loop. From the disassembly in listing 2.3, it is pretty clear that this rather simplistic explanation of the operation of the interpreter loop leaves out a lot of details. A few of these outstanding questions may include.



1. Where are the values such as that loaded by the `LOAD_FAST` instruction gotten from?
2. Where do arguments used as part of the instructions come from?
3. How are nested function and method calls managed?
4. How does the interpreter loop handle exceptions?

After the module's execution, the `Py_Main` function continues with the clean-up process. Just as `Py_Initialize` performs initialization during the interpreter startup, `Py_FinalizeEx` is invoked to do some clean-up work; this clean-up process involves waiting for threads to exit, calling any exit hooks, freeing up any memory allocated by the interpreter that is still in use, and so on, paving the way for the interpreter to exit.

The above is a high-level overview of the processes involved in executing a Python module. A lot of details are left out at this stage, but all will be revealed in subsequent chapters. We continue in the next chapter with a description of the compilation process.

3. Compiling Python Source Code

Although most people may not regard Python as a compiled language, it is one. During compilation, the interpreter generates executable bytecode from Python source code. However, Python's compilation process is a relatively simple one. It involves the following steps in order.

1. Parsing the source code into a parse tree.
2. Transforming the parse tree into an abstract syntax tree (AST).
3. Generating the symbol table.
4. Generating the code object from the AST. This step involves:
 1. Transforming the AST into a flow control graph, and
 2. Emitting a code object from the control flow graph.

Parsing source code into a parse tree and creating an AST from such a parse is a standard process and Python does not introduce any complicated nuances, so the focus of this chapter is on the transformation of an AST into a control flow graph and the emission of code object from the control flow graph. For anyone interested in parse tree and AST generation, the [dragon book](#)¹ provides an in-depth *tour de force* of both topics.

3.1 From Source To Parse Tree

The Python parser is an [LL\(1\)](#)² parser based on the description of such parsers laid out in the Dragon book. The `Grammar` module contains the Extended Backus-Naur Form (*EBNF*) grammar specification of the Python language. Listing 3.0 is a cross-section of this grammar.

Listing 3.0: A cross section of the Python BNF Grammar

```
1 stmt: simple_stmt | compound_stmt
2 simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
3 small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
4               import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
5 expr_stmt: testlist_star_expr (augassign (yield_expr|testlist) |
6                               ('=' (yield_expr|testlist_star_expr))*)|
7 testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
8 augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^='
9           | '<<=' | '>>=' | '**=' | '//=')
```

10

¹<https://www.amazon.co.uk/Compilers-Principles-Techniques-Alfred-Aho/dp/0201100886>

²https://en.wikipedia.org/wiki/LL_parser

```

11 del_stmt: 'del' exprlist
12 pass_stmt: 'pass'
13 flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt |
14     yield_stmt
15 break_stmt: 'break'
16 continue_stmt: 'continue'
17 return_stmt: 'return' [testlist]
18 yield_stmt: yield_expr
19 raise_stmt: 'raise' [test ['from' test]]
20 import_stmt: import_name | import_from
21 import_name: 'import' dotted_as_names
22 import_from: ('from' ('.' | '...')* dotted_name | ('.' | '...')+
23     'import' ('*' | '(' import_as_names ')' | import_as_names))
24 import_as_name: NAME ['as' NAME]
25 dotted_as_name: dotted_name ['as' NAME]
26 import_as_names: import_as_name (',' import_as_name)* [',']
27 dotted_as_names: dotted_as_name (',' dotted_as_name)*
28 dotted_name: NAME ('.' NAME)*
29 global_stmt: 'global' NAME (',' NAME)*
30 nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
31 assert_stmt: 'assert' test [',' test]
32
33 ...

```

The `PyParser_ParseFileObject` function in `Parser/parsetok.c` is the entry point for parsing any module passed to the interpreter at the command-line. This function invokes the `PyTokenizer_FromFile` function that is responsible for generating tokens from the supplied modules.

3.2 Python tokens

Python source code consists of tokens. For example, `return` is a keyword token; `2` is a literal numeric token. Tokenization, the splitting of source code into constituent tokens, is the first task during parsing. The tokens from this step fall into the following categories.

1. **identifiers:** These are names defined by a programmer. They include function names, variable names, class names, etc. These must conform to the rules of identifiers specified in the Python documentation.
2. **operators:** These are special symbols such as `+`, `*` that operate on data values, and produce results.
3. **delimiters:** This group of symbols serve to group expressions, provide punctuations, and assignment. Examples in this category include `(`, `)`, `{`, `}`, `=`, `*=` etc.

4. literals: These are symbols that provide a constant value for some type. We have the string and byte literals such as "Fred", b"Fred" and numeric literals which include integer literals such as 2, floating-point literal such as 1e100 and imaginary literals such as 10j.
5. comments: These are string literals that start with the hash symbol. Comment tokens always end at the end of the physical line.
6. NEWLINE: This is a unique token that denotes the end of a logical line.
7. INDENT and DEDENT: These token represent indentation levels that group compound statements.

A group of tokens delineated by the **NEWLINE** token makes up a logical line; hence we could say that a Python program consists of a sequence of *logical lines*. Each of these logical lines consists of several physical lines that are each terminated by an end-of-line sequence. Most times, logical lines map to physical lines, so we have a logical line delimited by end-of-line characters. These logical lines usually map to Python statements. Compound statements may span multiple physical lines; parenthesis, square brackets or curly braces around a statement implicitly joins the logical lines that make up such statement. The backslash character, on the other hand, is needed to join multiple logical lines explicitly.

Indentation also plays a central role in grouping Python statements. One of the lines in the Python grammar is thus suite: `simple_stmt | NEWLINE INDENT stmt+ DEDENT` so a crucial task of the tokenizer generating `indent` and `dedent` tokens that go into the parse tree. The tokenizer uses an algorithm similar to that in Listing 3.1 to generate these `INDENT` and `DEDENT` tokens.

Listing 3.1: Python indentation algorithm for generating INDENT and DEDENT tokens

```

1  Init the indent stack with the value 0.
2  For each logical line taking into consideration line-joining:
3      A. If the current line's indentation is greater than the
4          indentation at the top of the stack
5          1. Add the current line's indentation to the top of the stack.
6          2. Generate an INDENT token.
7      B. If the current line's indentation is less than the indentation
8          at the top of the stack
9          1. If there is no indentation level on the stack that matches the current li\
10 ne's indentation, report an error.
11          2. For each value at the top of the stack, that is unequal to the current li\
12 ne's indentation.
13              a. Remove the value from the top of the stack.
14              b. Generate a DEDENT token.
15      C. Tokenize the current line.
16  For every indentation on the stack except 0, produce a DEDENT token.

```

The `PyTokenizer_FromFile` function in the `Parser/tokenizer.c` scans the source file from left to right and top to bottom tokenizing the file's contents and then outputting a `tokenizer` structure.

Whitespaces characters other than terminators serve to delimit tokens but are not compulsory. In cases of ambiguity such as in 2+2, a token comprises the longest possible string that forms a legal token reading from left to right; in this example, the tokens are the literal 2, the operator + and the literal 2.

The tokenizer structure generated by the `PyTokenizer_FromFile` function gets passed to the `parsetok` function that attempts to build a parse tree according to the Python grammar of which Listing 3.0 is a subset. When the parser encounters a token that violates the Python grammar, it raises a `SyntaxError` exception. The `parser`³ module provides limited access to the parse tree of a block of Python code, and listing 3.2 is a basic demonstration.

Listing 3.2: Using the parser module to obtain the parse tree of python code

```
1  >>>code_str = """def hello_world():
2          return 'hello world'
3          """
4
5  >>> import parser
6  >>> from pprint import pprint
7  >>> st = parser.suite(code_str)
8  >>> pprint(parser.st2list(st))
9
10 [257,
11 [269,
12 [294,
13 [263,
14 [1, 'def'],
15 [1, 'hello_world'],
16 [264, [7, '('], [8, ')']],
17 [11, ':'],
18 [303,
19 [4, ''],
20 [5, ''],
21 [269,
22 [270,
23 [271,
24 [277,
25 [280,
26 [1, 'return'],
27 [330,
28 [304,
29 [308,
30 [309,
31 [310,
32 [311,
```

³<https://docs.python.org/3.6/library/parser.html#module-parse>

The `parser.suite(source)` call in listing 3.2 returns an intermediate representation of a parse tree (ST) object while the call to `parser.st2list` returns the parse tree represented by a Python list - each list represents a node of the parse tree. The first items in each list, the integer, identifies the production rule in the Python grammar responsible for that node.

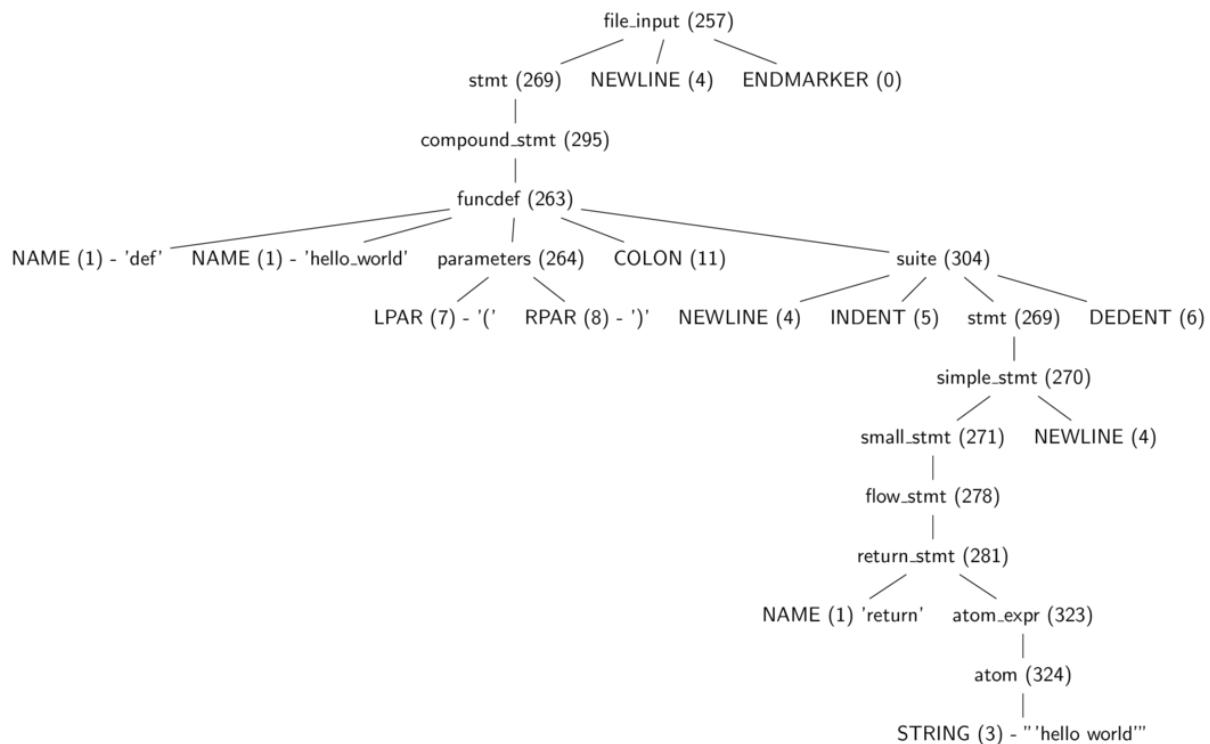


Figure 3.0: A parse tree for listing 3.2 (a function that returns the ‘hello world’ string)

Figure 3.0 is a tree diagram of the same parse tree from listing 3.2 with some tokens stripped away, and one can see more easily the part of the grammar each of the integer value represents. These production rules are all specified in the `Include/token.h` (terminals) and `Include/graminit.h` (terminals) header files.

Listing 3.3 from the `Include/node.h` shows the data structure that represents a parse tree in CPython. Each production rule is a *node* on the tree data structure. Each node has child nodes, and together the root node and its offspring nodes represent the parse tree.

Listing 3.3: The node data structure used in the CPython virtual machine

```
1 typedef struct _node {
2     short          n_type;
3     char           *n_str;
4     int            n_lineno;
5     int            n_col_offset;
6     int            n_nchildren;
7     struct _node  *n_child;
8 } node;
```

The macros for interacting with parse tree nodes are also in the `Include/node.h` file.

3.3 From Parse Tree To Abstract Syntax Tree

The parse tree is dense with information about Python's syntax, and all that information such as how lines are delimited is irrelevant for generating bytecode. This is where the abstract syntax tree (AST) comes in. The abstract syntax tree is a representation of the code that is independent of Python's syntax niceties. For example, a parse tree contains syntax constructs such as colon and NEWLINE nodes, as shown in figure 3.0, but the AST does not include such syntax construct as shown in listing 3.4. The transformation of the parse tree to the abstract syntax tree is the next step in the compilation pipeline.

Listing 3.4: Using the `ast` module to manipulate the AST of python source code

```

1 >>> import ast
2 >>> import pprint
3 >>> node = ast.parse(code_str, mode="exec")
4 >>> ast.dump(node)
5 ("Module(body=[FunctionDef(name='hello_world', args=arguments(args=[], "
6 'vararg=None, kwonlyargs=[], kw_defaults=[], kwarg=None, defaults=[]), '
7 "body=[Return(value=Str(s='hello world'))], decorator_list=[], "
8 'returns=None)])")

```

Python makes use of the Zephyr Abstract Syntax Definition Language (ASDL), and the ASDL definitions of the various Python constructs are in the file `Parser/Python.asdl` file. Listing 3.5 is a fragment of the *ASDL* definition of a Python statement.

Listing 3.5: A fragment of ASDL definition of a Python statement

```

1 stmt = FunctionDef(identifier name, arguments args,
2                     stmt* body, expr* decorator_list, expr? returns)
3     | AsyncFunctionDef(identifier name, arguments args,
4                         stmt* body, expr* decorator_list, expr? returns)
5
6     | ClassDef(identifier name,
7                 expr* bases,
8                 keyword* keywords,
9                 stmt* body,
10                expr* decorator_list)
11    | Return(expr? value)
12
13    | Delete(expr* targets)
14    | Assign(expr* targets, expr value)
15    | AugAssign(expr target, operator op, expr value)
16    -- 'simple' indicates that we annotate simple name without parens
17    | AnnAssign(expr target, expr annotation, expr? value, int simple)
18
19    -- use 'orelse' because else is a keyword in target languages
20    | For(expr target, expr iter, stmt* body, stmt* orelse)
21    | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
22    | While(expr test, stmt* body, stmt* orelse)
23    | If(expr test, stmt* body, stmt* orelse)
24    | With(withitem* items, stmt* body)
25    | AsyncWith(withitem* items, stmt* body)

```

In the CPython, C structures defined in `Include/Python-ast.h` represent the various AST nodes. The `Parser/asdl_c.py` module auto-generates generates this file from the AST *ASDL* definitions. Listing 3.6 is a fragment of the C code generated for a Python statement node.

Listing 3.6: A cross-section of an AST statement node data A> A> structure

```

1  struct _stmt {
2      enum _stmt_kind kind;
3      union {
4          struct {
5              identifier name;
6              arguments_ty args;
7              asdl_seq *body;
8              asdl_seq *decorator_list;
9              expr_ty returns;
10         } FunctionDef;
11
12         struct {
13             identifier name;
14             arguments_ty args;
15             asdl_seq *body;
16             asdl_seq *decorator_list;
17             expr_ty returns;
18         } AsyncFunctionDef;
19
20         struct {
21             identifier name;
22             asdl_seq *bases;
23             asdl_seq *keywords;
24             asdl_seq *body;
25             asdl_seq *decorator_list;
26         } ClassDef;
27         ...
28     }v;
29     int lineno;
30     int col_offset
31 }
```

The union type in listing 3.6 is a C type that can represent any of the types in the union.

The `PyAST_FromNode` function in the `Python/ast.c` calls `PyAST_FromNodeObject` also in `Python/ast.c` which walks the various parse tree nodes and generates AST nodes accordingly using functions defined in `Python/ast.c`. The heart of this function is a large switch statement that calls node specific functions on each node type. For example, the code responsible for generating the AST node for an

if expression is in listing 3.7.

Listing 3.7: Function for generating an AST node for an if expression

```

1 static expr_ty
2 ast_for_ifexpr(struct compiling *c, const node *n)
3 {
4     /* test: or_test 'if' or_test 'else' test */
5     expr_ty expression, body, orelse;
6
7     assert(NCH(n) == 5);
8     body = ast_for_expr(c, CHILD(n, 0));
9     if (!body)
10         return NULL;
11     expression = ast_for_expr(c, CHILD(n, 2));
12     if (!expression)
13         return NULL;
14     orelse = ast_for_expr(c, CHILD(n, 4));
15     if (!orelse)
16         return NULL;
17     return IfExp(expression, body, orelse, LINENO(n), n->n_col_offset,
18                 c->c_arena);
19 }
```



We will see that all other interactions with the AST make use of a similar pattern where node specific functions recursively walk the AST from left to right with the help of macros in `Include/asdl.h` and node-specific functions.

Listing 3.8: A simple python function

```

1 def fizzbuzz(n):
2     if n % 3 == 0 and n % 5 == 0:
3         return 'FizzBuzz'
4     elif n % 3 == 0:
5         return 'Fizz'
6     elif n % 5 == 0:
7         return 'Buzz'
8     else:
9         return str(n)
```

Take the code in Listing 3.8, for example, the transformation of its parse tree to an AST will result in an AST similar to figure 3.1.

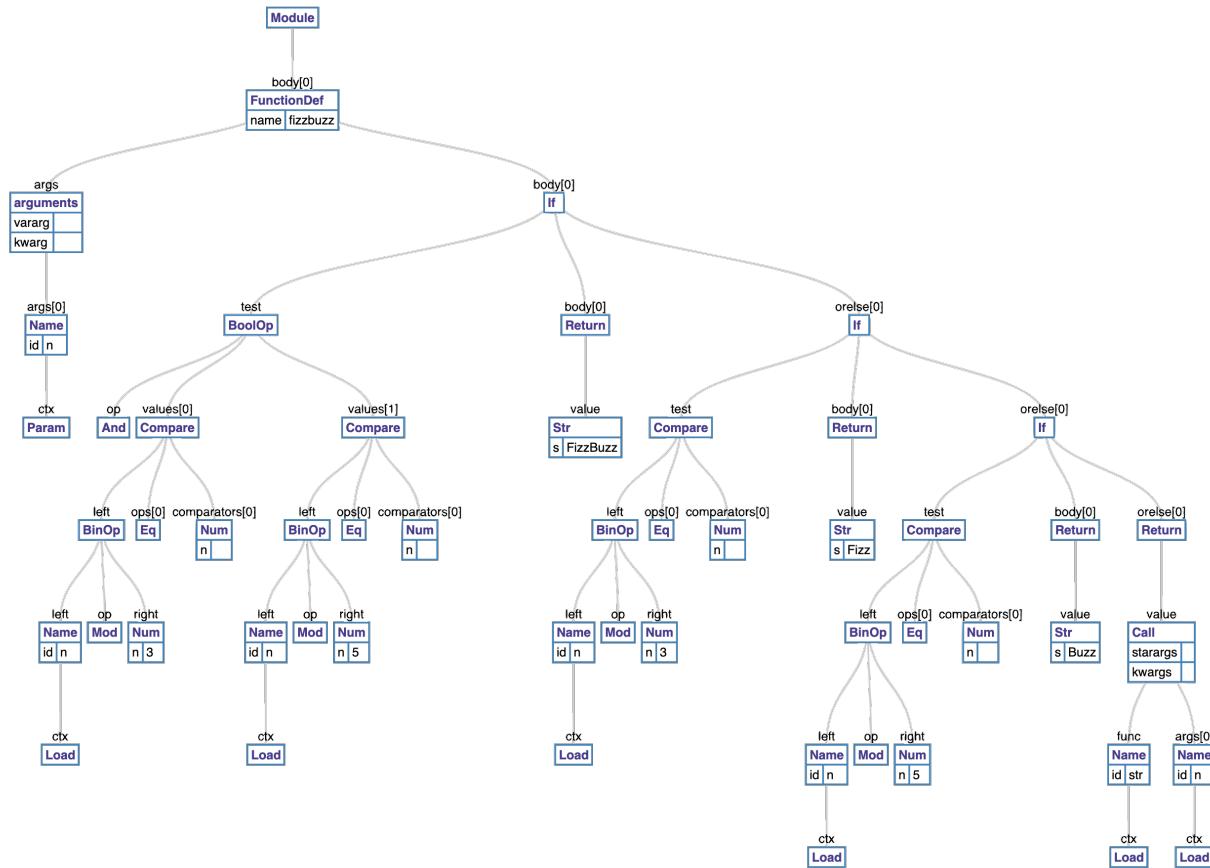


Figure 3.1: An AST for listing 3.2

The `ast` module bundled with the Python interpreter provides us with the ability to manipulate a Python AST. Tools such as `codegen`⁴ can take an AST representation in Python and output the corresponding Python source code.

With the AST generated, the next step is creating the symbol table.

3.4 Building The Symbol Table

The symbol table, as the name suggests, is a collection of symbols and their use context within a code block. Building the symbol table involves analyzing and assigning scoping to the names in a code block.

Names and Binding

In Python, objects are referenced by *names*. *names* are analogous to *but not exactly* variables in C++ and Java.

⁴<https://pypi.python.org/pypi/codegen/1.0>

```
>>> x = 5
```

In the above example, `x` is a name that references the object, 5. The process of *assigning* a reference to 5 to `x` is called *binding*. A binding causes a name to be associated with an object in the innermost scope of the currently executing program. Bindings may occur during several instances such as during variable assignment or function or method call when the supplied parameter is bound to the argument. It is important to note that names are just symbols and they have no *type* associated with them; **names are only references to objects that have types**.

Code Blocks

Code blocks are central to Python program, and a understanding of them is crucial. A code block is a piece of program code that executes as a single unit in Python. Modules, functions, and classes are all examples of code blocks. Commands typed in interactively at the REPL, script commands run with the `-c` option are also code blocks. A code block has several namespaces associated with it. For example, a module code block has access to the `global` namespace while a function code block has access to the `local` as well as the `global` namespaces.

Namespaces

A * namespace* defines a context in which a given set of names is bound to objects. Namespaces are implemented using dictionary mappings. The `builtin` namespace is an example of a namespace that contains all the built-in functions and this can be accessed by entering `__builtins__.__dict__` at the terminal (the result is of a considerable amount). The interpreter has access to multiple namespaces including *the global namespace*, *the builtin namespace* and *the local namespace*. These namespaces are created at different times and have different lifetimes. For example, invoking a function will create a new *local* namespace and the interpreter drops that namespace when the function exits or returns. The *global* namespace is created at the start of the execution of a module and all names defined in this namespace are available module-wide while the *built-in* namespace comes into existence when the interpreter is invoked and contains all the builtin names. These three namespaces are the main namespaces available to the interpreter.

Scopes

A scope is an area of a program in which a set of name bindings (namespaces) is visible and directly accessible without using any dot notation. At runtime, the following scopes may be available.

1. Innermost scope with local names,
2. The scope of enclosing functions if any (this is applicable for nested functions),
3. The current module's `globals` scope, and
4. The scope containing the `builtin` namespace.

When a name used, the interpreter searches the scopes' namespaces in the order listed above; if the interpreter does not find that name, it raises an exception. Python supports *static scoping* also known as *lexical scoping*; this means that the visibility of a set of name bindings can be inferred by only inspecting the program text.

Note

Python has a quirky scoping rule that prohibits modifying a reference to an object in the *global* scope from a local scope; such an attempt will throw an `UnboundLocalError` exception. In order to modify an object from the global scope within a local scope, the `global` keyword is used with the object name before attempting any modification. The snippet in Listing 3.9 illustrates this.

Listing 3.9: Attempting to modify a global variable from a function

```
>>> a = 1
>>> def inc_a(): a += 2
...
>>> inc_a()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in inc_a
UnboundLocalError: local variable 'a' referenced before assignment
```

The `global` statement is used to modify the object from the global scope, as shown in listing 3.10.

Listing 3.10: Using the global keyword to modify a global variable from a function

```
>>> a = 1
>>> def inc_a():
...     global a
...     a += 1
...
>>> inc_a()
>>> a
2
```

Python also has the `nonlocal` keyword used when there is a need to modify a variable bound in an outer non-global scope from an inner scope. This keyword is handy when working with nested functions (also referred to as closures). The snippet in listing 3.11 that defines a simple counter object illustrates the usage of the `nonlocal` keyword.

Listing 3.11: Creating blocks from an AST

```
>>> def make_counter():
...     count = 0
...     def counter():
...         nonlocal count #capture count binding from enclosing not global scope
...         count += 1
...         return count
...     return counter
...
>>> counter_1 = make_counter()
>>> counter_2 = make_counter()
>>> counter_1()
1
```

```

>>> counter_1()
2
>>> counter_2()
1
>>> counter_2()
2

```

Symbol table data structures

Two data structures that are central to generating a symbol table are:

1. The `symtable` data structure.
2. The `symtable_entry` data structure.

Listing 3.13 shows the `symtable` data structure. This data structure is a table composed of entries that hold information on A> the names used in a module's code blocks.

Listing 3.13: The `symtable` data structure

```

1  struct symtable {
2      PyObject *st_filename;           /* name of file being compiled */
3      struct _symtable_entry *st_cur; /* current symbol table entry */
4      struct _symtable_entry *st_top; /* symbol table entry for module */
5      PyObject *st_blocks;           /* dict: map AST node addresses
6                                         * to symbol table entries */
7      PyObject *st_stack;           /* list: stack of namespace info */
8      PyObject *st_global;          /* borrowed ref to
9                                         * st_top->ste_symbols */
10     int st_nbblocks;              /* number of blocks used. kept for
11                                         * consistency with the corresponding
12                                         * compiler structure */
13     PyObject *st_private;          /* name of current class or NULL */
14     PyFutureFeatures *st_future;  /* module's future features that
15                                         * affect the symbol table */
16     int recursion_depth;          /* current recursion depth */
17     int recursion_limit;          /* recursion limit */
18 };

```

A module may contain many code blocks - such as multiple function definitions - and the `st_blocks` field maps each of the A> code blocks to a symbol table entry. The `st_top` is a symbol table entry for the module (recall that a module is also a code block), so it contains the names defined in the modules' global namespace. The `st_cur` is the symbol table entry for the current code block. Figure 3.2 is a graphical illustration of this structure.

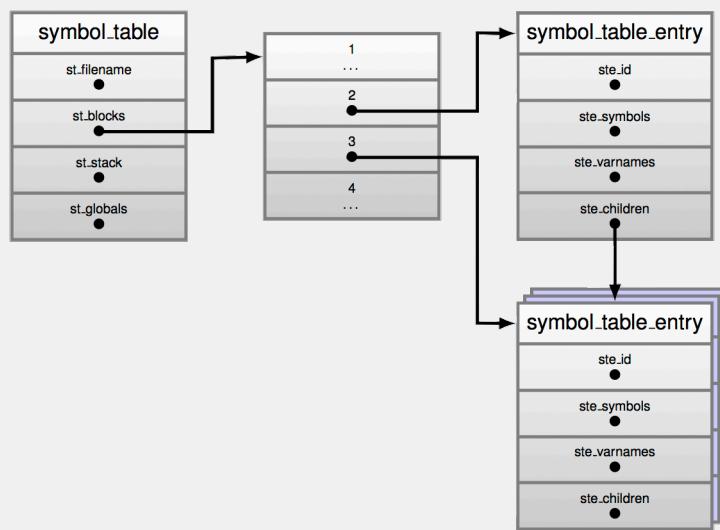


Figure 3.2: A Symbol table and symbol table entries.

Listing 3.14 is the definition of the `_symtable_entry` data structure; this is also in `Include/symtable.h`.

Listing 3.14: The `_symtable_entry` data structure

```

1  typedef struct _symtable_entry {
2      PyObject_HEAD
3      PyObject *ste_id;           /* int: key in ste_table->st_blocks */
4      PyObject *ste_symbols;      /* dict: variable names to flags */
5      PyObject *ste_name;         /* string: name of current block */
6      PyObject *ste_varnames;     /* list of function parameters */
7      PyObject *ste_children;     /* list of child blocks */
8      PyObject *ste_directives;   /* locations of global and nonlocal
9                                statements */
10     _Py_block_ty ste_type;      /* module, class, or function */
11     int ste_nested;            /* true if block is nested */
12     unsigned ste_free : 1;       /* true if block has free variables */
13     unsigned ste_child_free : 1; /* true if a child block has free
14                                vars including free refs to globals */
15     unsigned ste_generator : 1;   /* true if namespace is a generator */
16     unsigned ste_varargs : 1;     /* true if block has varargs */
17     unsigned ste_varkeywords : 1; /* true if block has varkeywords */
18     unsigned ste_returns_value : 1; /* true if namespace uses return with
19                                an argument */
20     unsigned ste_needs_class_closure : 1; /* for class scopes, true if a
21                                closure over __class__
22                                should be created */
23     int ste_lineno;             /* first line of block */
24     int ste_col_offset;         /* offset of first line of block */
25     int ste_opt_lineno;         /* lineno of last exec or import */

```

```

26         int ste_opt_col_offset; /* offset of last exec or import */
27         int ste_tmpname;        /* counter for listcomp temp vars */
28         struct symtable *ste_table;
29     } PySTEntryObject;

```

The comments in Listing 3.14 explain the function of each field. The `ste_symbols` field is a mapping of symbols in the code block to context flags. The flags are numeric values that provide information on the use context of the given symbol. For example, a symbol may be a function argument or a global statement definition. Listing 3.15 are some examples of the flags in `Include/symtable.h`.

Listing 3.15: Flags that specify context of a name definition

```

1     /* Flags for def-use information */
2     #define DEF_GLOBAL 1           /* global stmt */
3     #define DEF_LOCAL 2           /* assignment in code block */
4     #define DEF_PARAM 2<<1       /* formal parameter */
5     #define DEF_NONLOCAL 2<<2     /* nonlocal stmt */
6     #define DEF_FREE 2<<4        /* name used but not defined in
7                                         nested block */

```

The `PySymtable_BuildObject` function in `Python/compile.c` walks the AST to create the symbol table. This is a two-step process summarized in listing 3.12.

Listing 3.12: Creating a symbol table from an AST

```

For each node in a given AST
    If the node is the start of a code block:
        1. Create a new symbol table entry and set the current symbol table to this \
value.
        2. Push the new symbol table to st_stack.
        3. Add the new symbol table to the list of children of the previous symbol t\
able.
        4. Update the current symbol table to this new symbol table
        5. For all nodes in the code block nodes:
            a. recursively visit each node using the "symtable_visit_XXX"
                functions where "XXX" is a node type.
        6. Exit the code block by removing the current symbol table entry from the s\
tack.
        7. Pop the next symbol table entry from the stack and set the
            current symbol table entry to this popped value
    else:
        recursively visit the node and sub-nodes.

```

First, we visit each node of the AST to build a collection of symbols used. After the first pass, the symbol table entries contain all names that have been used within the module, but it does not have contextual information about such names. For example, the interpreter cannot tell if a given variable is a global, local, or free variable. The `symtable_analyze` function in the `Parser/symtable.c` handles the second phase. In this phase, the algorithm assigns scopes (local, global, or free) to the symbols gathered from the first pass. The comments in the `Parser/symtable.c` are quite informative and are paraphrased below to provide some insight into the second phase of the symbol table construction process.

The symbol table requires two passes to determine the scope of each name. The first pass collects raw facts from the AST via the `symtable_visit_*` functions while the second pass analyzes these facts during a pass over the `PySTEntryObjects` created during pass 1. During the second pass, the parent passes the set of all name bindings visible to its children when it enters a function. These bindings determine if nonlocal variables are free or implicit globals. Names which are explicitly declared nonlocal must exist in this set of visible names - if they do not, the interpreter raises a syntax error. After the local analysis, it analyzes each of its child blocks using an updated set of name bindings.

There are also two kinds of global variables, implicit and explicit. An explicit global is declared with the `global` statement. An implicit global is a free variable for which the compiler has found no binding in an enclosing function scope. The implicit global is either a global or a builtin.

Python's module and class blocks use the `xxx_NAME` opcodes to handle these names to implement slightly odd semantics. In such a block, the name is treated as global until it is assigned a value; then it is treated like a local.

The children update the free variable set. If a child adds a variable to the set of free variables, then such variable is marked as a cell. The function object defined must provide runtime storage for the variable that may outlive the function's frame. Cell variables are removed from the free set before the `analyze` function returns to its parent.

For example, a symbol table for a module with content in listing 3.16 will contain three symbol table entries.

Listing 3.16: A simple function

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter
```

The first entry is that of the enclosing module, and it will have `make_counter` defined with a local scope. The next symbol table entry will be that of function `make_counter`, and this will have the

count and counter names marked as local. The final symbol table entry will be that of the nested counter function. This entry will have the count variable marked as `free`. One thing to note is that although `make_counter` has a local scope in the symbol table entry for the module block, it is globally defined in the module code block because the `*st_global` field of the symbol table points to the `*st_top` symbol table entry which is that of the enclosing module.

3.5 From AST To Code Objects

After generating the symbol table, the next step is creating code objects. The functions for this step are in the `Python/compile.c` module. First, they convert the AST into basic blocks of Python byte code instructions. Basic blocks are blocks of code that have a single entry but can have multiple exits. The algorithm here uses a pattern similar to that used to generate the symbol table. Functions named `compiler_visit_xx`, where `xx` is the node type, recursively visit each node of the AST emitting bytecode instructions in the process. We see some examples of these functions in the sections that follow. The blocks of bytecode here implicitly represent a graph, the control flow graph. This graph shows the potential code execution paths. In the second step, the algorithm flattens the control flow graph using a post-order depth-first search transversal. After, the jump offsets are calculated and used as instruction arguments for bytecode `jmp` instructions. The bytecode instructions are then used to create a code object.

The compiler data structure

Figure 3.3 shows the relationship between the primary data structures used in the process of generating the fundamental blocks that make up the control flow graph.

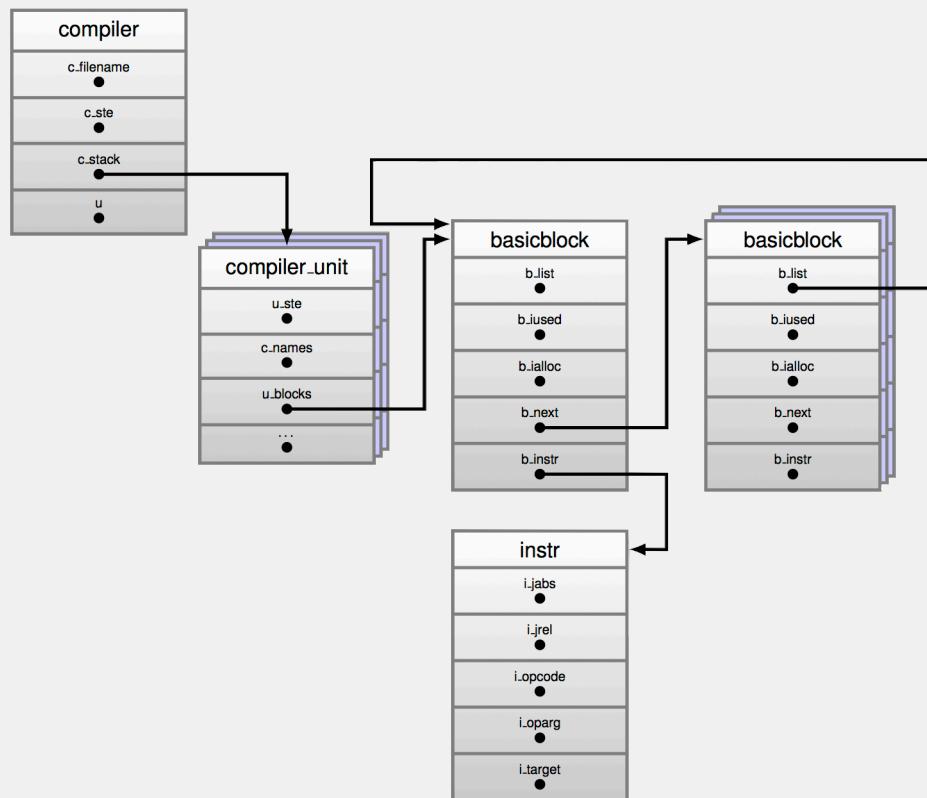


Figure 3.3: The four major data structures used in generating a code object.

At the very top level is the compiler data structure, which captures the global compilation process for a module. Listing 3.17 is a definition of this data structure.

Listing 3.17: The compiler data structure

```

1  struct compiler {
2      PyObject *c_filename;
3      struct symtable *c_st;
4      PyFutureFeatures *c_future; /* pointer to module's __future__ */
5      PyCompilerFlags *c_flags;
6
7      int c_optimize;           /* optimization level */
8      int c_interactive;       /* true if in interactive mode */
9      int c_nestlevel;
10
11     struct compiler_unit *u; /* compiler state for current block */

```

```

12         PyObject *c_stack;           /* Python list holding compiler_unit
13                               ptrs */
14         PyArena *c_arena;           /* pointer to memory allocation arena */
15     };

```

The fields that are of interest to us here are the following.

1. `*c_st`: a reference to the symbol table generated in the previous section.
2. `*u`: a reference to a compiler unit data structure. This encapsulates information needed for working with a code block. This field points to the compiler unit for the current code block that is being compiled.
3. `*c_stack`: a reference to a stack of `compiler_unit` data structures. When a code block is composed of multiple code blocks, this field manages the saving and restoration of `compiler_unit` data structures as the compiler encounters new blocks. When the compiler enters a new code block - it creates a new scope - then the `compiler_enter_scope()` pushes the current `compiler_unit` - `*u` - onto the stack - `*c_stack`, creates a new `compiler_unit` object, and sets it as the current state for that new block encountered. When the compiler exits the block, the `*c_stack` is popped off the stack accordingly to restore the state.

The compiler initializes a `compiler` data structure to compile every Python module; as the AST generated for the module is walked, the compiler creates a `compiler_unit` data structure for each code block that it encounters within the AST.

The `compiler_unit` data structure

The `compiler_unit` data structure shown in Listing 3.18 captures the information needed to generate the required byte code instructions for a code block. We discuss most of the fields defined in the `compiler_unit` when we look at code objects.

Listing 3.18: The `compiler_unit` data structure

```

1  struct compiler_unit {
2     PySTEntryObject *u_st;
3
4     PyObject *u_name;
5     PyObject *u_qualname; /* dot-separated qualified name (lazy) */
6     int u_scope_type;
7
8     /* The following fields are dicts that map objects to the index of them in
9     n co_XXX. The index is an argument for opcodes that refer to those collections.
10    */
11    PyObject *u_consts;    /* all constants */
12    PyObject *u_names;    /* all names */
13    PyObject *u_varnames; /* local variables */
14    PyObject *u_cellvars; /* cell variables */
15    PyObject *u_freevars; /* free variables */

```

```

17         PyObject *u_private;           /* for private name mangling */
18
19         Py_ssize_t u_argcount;        /* number of arguments for block */
20         Py_ssize_t u_kwonlyargcount; /* number of keyword only arguments
21                                         for block */
22         /* Pointer to the most recently allocated block. By following b_list
23         members, you can reach all early allocated blocks. */
24         basicblock *u_blocks;
25         basicblock *u_curblock; /* pointer to current block */
26
27         int u_nfblocks;
28         struct fblockinfo u_fblock[CO_MAXBLOCKS];
29
30         int u_firstlineno; /* the first lineno of the block */
31         int u_lineno;        /* the lineno for the current stmt */
32         int u_col_offset;    /* the offset of the current stmt */
33         int u_lineno_set;   /* boolean to indicate whether instr
34                             has been generated with current lineno */
35     };

```

The `u_blocks` and `u_curblock` fields reference the basic blocks that make up the compiled code block. The `*u_st` field refers to a symbol table entry for the code block the compiler is processing. The rest of the fields have pretty self-explanatory names. The compiler walks the different nodes that make up the code block and depending on whether a given node type begins a basic block or not, a basic block containing the nodes instructions is created or instructions for the node get added to an existing basic block. Node types that may begin a new basic block include but are not limited to the following.

1. Function nodes.
2. Jump targets.
3. Exception handlers.
4. Boolean operations, etc.

Basic blocks

The basic block is central to generating code objects. A basic block is a sequence of instructions that has one entry point but multiple exit points. Listing 3.19 is the definition of the `basic_block` data structure.

Listing 3.19: The `basicblock_data` structure

```

1 typedef struct basicblock_ {
2     /* Each basicblock in a compilation unit is linked via b_list in the
3     reverse order that the block are allocated. b_list points to the next
4     block, not to be confused with b_next, which is next by control flow. */
5     struct basicblock_ *b_list;
6     /* number of instructions used */
7     int b_iused;
8     /* length of instruction array (b_instr) */
9     int b_ialloc;
10    /* pointer to an array of instructions, initially NULL */
11    struct instr *b_instr;
12    /* If b_next is non-NULL, it is a pointer to the next
13    block reached by normal control flow. */
14    struct basicblock_ *b_next;
15    /* b_seen is used to perform a DFS of basicblocks. */
16    unsigned b_seen : 1;
17    /* b_return is true if a RETURN_VALUE opcode is inserted. */
18    unsigned b_return : 1;
19    /* depth of stack upon entry of block, computed by stackdepth() */
20    int b_startdepth;
21    /* instruction offset for block, computed by assemble_jump_offsets() */
22    int b_offset;
23 } basicblock;

```

The interesting fields here are `*b_list` that is a linked list of all basic blocks allocated during the compilation process, `*b_instr` which is an array of instructions within the basic block and `*b_next` which is the next basic block reached by normal control flow execution. Each instruction has a structure shown in Listing 3.20 that holds a bytecode instruction. These bytecode instructions are in the `Include/opcode.h` header file.

Listing 3.20: The `instr` data structure

```

1 struct instr {
2     unsigned i_jabs : 1;
3     unsigned i_jrel : 1;
4     unsigned char i_opcode;
5     int i_oparg;
6     struct basicblock_ *i_target; /* target block (if jump instruction) */
7     int i_lineno;
8 };

```

To illustrate how the interpreter creates these basic blocks, we use the function in Listing 3.11. Compiling its AST shown in figure 3.2 into a CFG results in the graph similar to that in figure

3.4 - this shows only blocks with instructions. An inspection of this graph provides some intuition behind the basic blocks. Some basic blocks have a single entry point, but others have multiple exits. These blocks are described in more detail next.

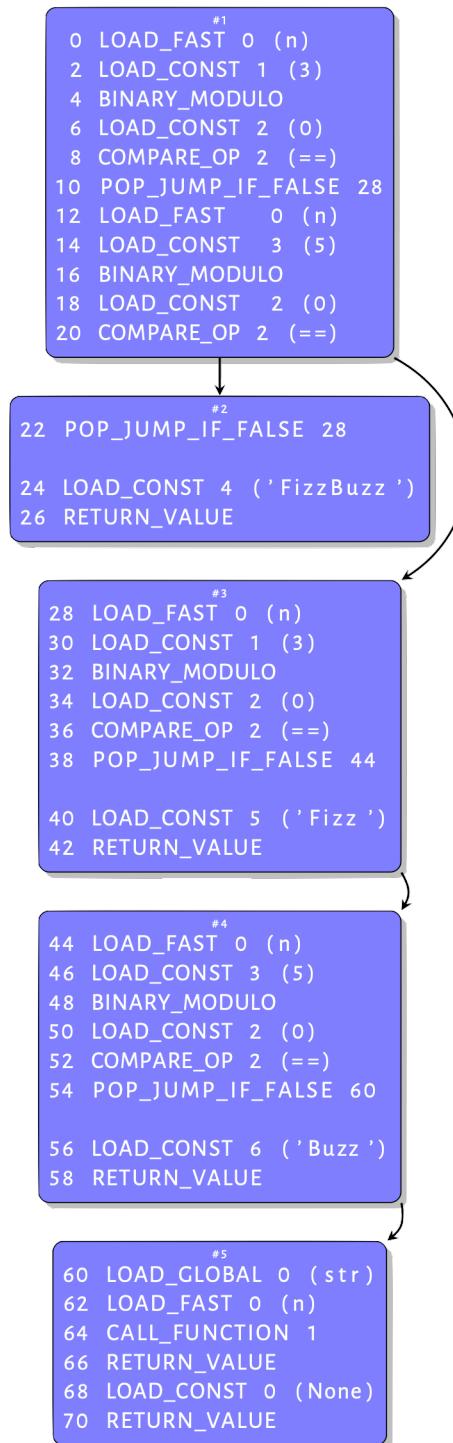


Figure 3.4: Control flow graph for the fizzbuzz function in listing 3.11.

Listing 3.21: Compiling an `if` statement

```

1 static int
2 compiler_if(struct compiler *c, stmt_ty s)
3 {
4     basicblock *end, *next;
5     int constant;
6     assert(s->kind == If_kind);
7     end = compiler_new_block(c);
8     if (end == NULL)
9         return 0;
10
11    constant = expr_constant(c, s->v.If.test);
12    /* constant = 0: "if 0"
13     * constant = 1: "if 1", "if 2", ...
14     * constant = -1: rest */
15    if (constant == 0) {
16        if (s->v.If.orelse)
17            VISIT_SEQ(c, stmt, s->v.If.orelse);
18    } else if (constant == 1) {
19        VISIT_SEQ(c, stmt, s->v.If.body);
20    } else {
21        if (asdl_seq_LEN(s->v.If.orelse)) {
22            next = compiler_new_block(c);
23            if (next == NULL)
24                return 0;
25        }
26        else
27            next = end;
28        VISIT(c, expr, s->v.If.test);
29        ADDOP_JABS(c, POP_JUMP_IF_FALSE, next);
30        VISIT_SEQ(c, stmt, s->v.If.body);
31        if (asdl_seq_LEN(s->v.If.orelse)) {
32            ADDOP_JREL(c, JUMP_FORWARD, end);
33            compiler_use_next_block(c, next);
34            VISIT_SEQ(c, stmt, s->v.If.orelse);
35        }
36    }
37    compiler_use_next_block(c, end);
38    return 1;
39 }

```

The body of the function in Listing 3.13 is an `if` statement as is visible in figure 3.2. The snippet

in Listing 3.21 is the function that compiles an `if` statement AST node into basic blocks. When this function compiles our example `if` statement node, the `else` statement on line 20 is executed. First, it creates a new basic block for an `else` node if such exists. Then, it visits the guard clause of the `if` statement node. What we have in the function in Listing 3.11 is interesting because the guard clause is a boolean expression which can trigger a jump during execution. Listing 3.22 is the function that compiles a boolean expression.

Listing 3.22: Compiling a boolean statement

```

1  static int compiler_boolop(struct compiler *c, expr_ty e)
2  {
3      basicblock *end;
4      int jumpi;
5      Py_ssize_t i, n;
6      asdl_seq *s;
7
8      assert(e->kind == BoolOp_kind);
9      if (e->v.BoolOp.op == And)
10         jumpi = JUMP_IF_FALSE_OR_POP;
11     else
12         jumpi = JUMP_IF_TRUE_OR_POP;
13     end = compiler_new_block(c);
14     if (end == NULL)
15         return 0;
16     s = e->v.BoolOp.values;
17     n = asdl_seq_LEN(s) - 1;
18     assert(n >= 0);
19     for (i = 0; i < n; ++i) {
20         VISIT(c, expr, (expr_ty)asdl_seq_GET(s, i));
21         ADDOP_JABS(c, jumpi, end);
22     }
23     VISIT(c, expr, (expr_ty)asdl_seq_GET(s, n));
24     compiler_use_next_block(c, end);
25     return 1;
26 }
```

The code up to the loop at line 20 is straight forward. In the loop, the compiler visits each expression, and after each visit, it adds a `jump`. This is because of the short circuit evaluation used by Python. It means that when a boolean operation such as an `AND` evaluates to `false`, the interpreter ignores the other expressions and performs a `jump` to continue execution. The compiler knows where to `jump` to if need be because the following instructions go into a new basic block - the use of `compiler_use_next_block` enforces this. So we have two blocks now. After visiting the test, the `compiler_if` function adds a `jump` instruction for the `if` statement, then compiles the body of the `if` statement. Recall that after visiting the boolean expression, the compiler created a new basic block. This block

contains the jump and instructions for body of the `if` statement, a simple return in this case. The target of this jump is the next block that will hold the `elif` arm of the `if` statement. The next step is to compile the `elif` component of the `if` statement but before this, the compiler calls the `compiler_use_next_block` function to activate the `next` block. The `orElse` arm is just another `if` statement, so the `compiler_if` function gets called again. This time around the test of the `if` is a compare operation. This is a single comparison, so there are no jumps involved and no new blocks, so the interpreter emits byte code for comparing values and returns to compile the body of the `if` statement. The same process continues for the last `orElse` arm resulting in the CFG in figure 3.3.

Figure 3.3 shows that the `fizzbuzz` function can exit block 1 in two ways. The first is via serial execution of all the instructions in block 1 then continuing in block 2. The other is via the `jump` instruction after the first compare operation. The target of this jump is block 3, but an executing code object knows nothing of basic blocks – the code object has a stream of bytecodes that are indexed with offsets. We have to provide the jump instructions with the offset into the bytecode instruction stream of the jump targets.

Assembling the basic blocks

The `assemble` function in `Python/compile.c` linearizes the CFG and creates the code object from the linearized CFG. It does so by computing the instruction offset for jump targets and using these as arguments to the jump instructions.

The `assembler` data structure in listing 3.23 plays a vital role during assembly of bytecode.

Listing 3.23: Calculating bytecode offsets

```

1 struct assembler {
2     PyObject *a_bytecode; /* string containing bytecode */
3     int a_offset;          /* offset into bytecode */
4     int a_nblocks;         /* number of reachable blocks */
5     basicblock **a_postorder; /* list of blocks in dfs postorder */
6     PyObject *a_lnotab;    /* string containing lnotab */
7     int a_lnotab_off;     /* offset into lnotab */
8     int a_lineno;          /* last lineno of emitted instruction */
9     int a_lineno_off;      /* bytecode offset of last lineno */
10    };

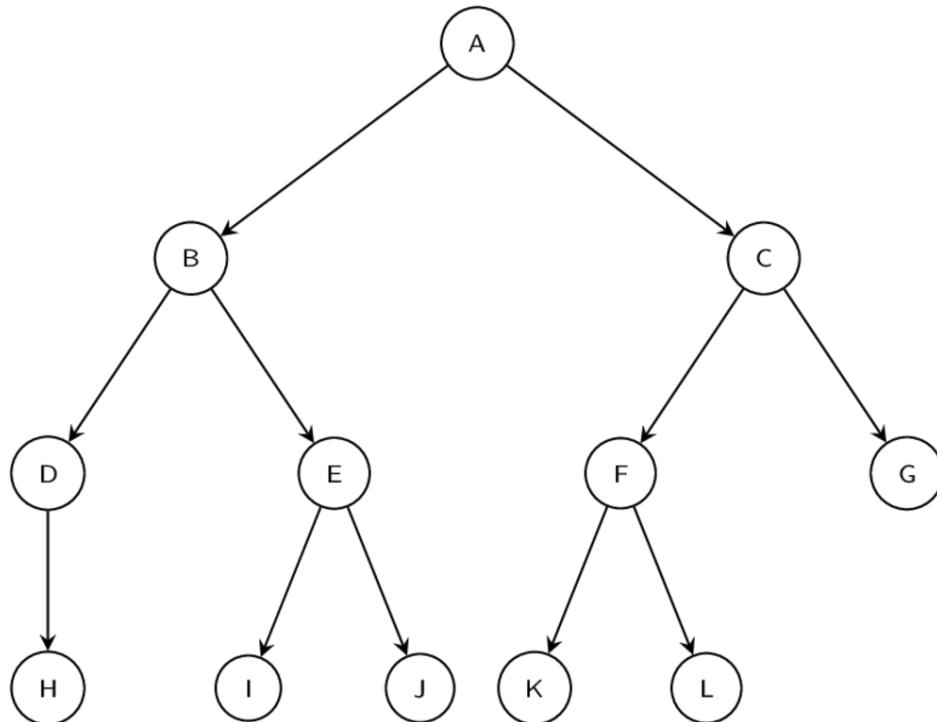
```

The `assembler` data structure holds among other things the `a_postorder` array of basic blocks that contains the basic blocks in post order, `a_bytecode` contains the bytecode string generated and `a_nblocks` is the number of basic blocks within the code block. As we go through this stage of the process, you will see how these all come into play.

First, the `assemble` function, in this case, adds instructions for a `return None` statement since the last

statement of the function is not a RETURN statement - now you know why you can define methods without adding a RETURN statement. Next, it flattens the CFG using a *post-order* depth-first traversal - the post-order traversal visits the children of a node before visiting the node itself. The assembler data structure holds the flattened graph, [block 5, block 4, block 3, block 2, block 1], in the `a_postorder` array for further processing. Next, it computes the instruction offsets and uses those as targets for the bytecode jump instructions. The `assemble_jump_offsets` function in listing 3.24 handles this.

Graph Traversal



In a post-order depth-first traversal of a graph, we recursively visit the left child node of the graph followed by the right child node of the graph then the node itself. In our graph in figure 3.45 when the graph is flattened using a post-order traversal, the order of the nodes is H → D → I → J → E → B → K → L → F → G → C → A. This is in contrast to an pre-order traversal that produces A → B → D → H → E → I → J → C → F → K → L → G or a in-order traversal that produces H → D → B → I → E → J → A → K → L → F → C → G

The `assemble_jump_offsets` function in Listing 3.24 is relatively straightforward. In the `for ... loop` at line 10, it computes the offset into the instruction stream for every instruction (akin to an array index). In the next `for ... loop` at line 17, it uses the computed offsets as arguments to the jump instructions distinguishing between absolute and relative jumps.

Listing 3.24: Calculating bytecode offsets

```

1 static void assemble_jump_offsets(struct assembler *a, struct compiler *c){
2     basicblock *b;
3     int bsize, totsize, extended_arg_recompile;
4     int i;
5
6     /* Compute the size of each block and fixup jump args.
7      Replace block pointer with position in bytecode. */
8     do {
9         totsize = 0;
10        for (i = a->a_nbblocks - 1; i >= 0; i--) {
11            b = a->a_postorder[i];
12            bsize = blocksize(b);
13            b->b_offset = totsize;
14            totsize += bsize;
15        }
16        extended_arg_recompile = 0;
17        for (b = c->u->u_blocks; b != NULL; b = b->b_list) {
18            bsize = b->b_offset;
19            for (i = 0; i < b->b_iused; i++) {
20                struct instr *instr = &b->b_instr[i];
21                int isize = instrsize(instr->i_oparg);
22                /* Relative jumps are computed relative to
23                 the instruction pointer after fetching
24                 the jump instruction.
25
26                */
27                bsize += isize;
28                if (instr->i_jabs || instr->i_jrel) {
29                    instr->i_oparg = instr->i_target->b_offset;
30                    if (instr->i_jrel) {
31                        instr->i_oparg -= bsize;
32                    }
33                    instr->i_oparg *= sizeof(_Py_CODEUNIT);
34                    if (instrsize(instr->i_oparg) != isize) {
35                        extended_arg_recompile = 1;
36                    }
37                }
38            }
39        } while (extended_arg_recompile);
40    }

```

With instructions offset calculated and jump offsets assembled, the compiler emits instructions contained in the flattened graph in reverse post-order from the traversal. The reverse post order is a topological sorting of the CFG. This means for every edge from vertex u to vertex v , u comes before v in the sorting order. This is obvious; we want a node that jumps to another node to always come before that jump target. After emitting the bytecode instructions, the compiler creates code objects for each code block using the emitted bytecode and information contained in the symbol table. The generated code object is returned to the calling function marking the end of the compilation process.

4. Python Objects

In this chapter, we look at the Python objects and their implementation in the CPython virtual machine. This is central to understanding the Python virtual machine's internals. Most of the source referenced in this chapter is available in the `Include/` and `Objects/` directories. Unsurprisingly, the implementation of the Python object system is quite complex, so we try to avoid getting bogged down in the gory details of the C implementation. To kick this off, we start by looking at the `PyObject` structure - the workhorse of the Python object system.

4.1 PyObject

A cursory inspection of the CPython source code reveals the ubiquity of the `PyObject` structure. As we will see later on in this treatise, all the value stack objects used by the interpreter during evaluation are `PyObject`s. For want of a better term, we refer to this as the *superclass* of all Python objects. Values are never declared as `PyObject` but a pointer to any object can be cast to a `PyObject`. In layman's term, any object can be treated as a `PyObject` structure because the initial segment of all objects is a `PyObject` structure.



A word on c structs.

When we say “values are never declared as a `PyObject` but a pointer to any object can be cast to a `PyObject`”, we refer to an implementation detail of the C programming language and how it interprets data at memory locations. C structs used to represent Python objects are just groups of bytes which we can interpret in any manner which we choose to. For example, a struct, `test`, may be composed of 5 `short` values each 2 bytes in size and summing up to 10 bytes. In C, given a reference to ten bytes, we can interpret those ten bytes as `test` struct composed of 5 `short` values regardless of whether the 10 bytes were defined as a `test` struct - however the output when you try to access the fields of the struct maybe gibberish. This means that given n bytes of data that represent a Python object where n is greater than the size of a `PyObject`, we can interpret the first n bytes as a `PyObject`.

Listing 4.0 is a definition of the `PyObject` structure. This structure is composed of several fields that must be filled for a value to be treated as an object.

Listing 4.0: PyObject definition

```

1  typedef struct _object {
2      _PyObject_HEAD_EXTRA
3      Py_ssize_t ob_refcnt;
4      struct _typeobject *ob_type;
5  } PyObject;

```

The `_PyObject_HEAD_EXTRA` when present is a C macro that defines fields that point to the previously allocated object and the next object, thus forming an implicit doubly-linked list of all live objects. The `ob_refcnt` field is for memory management, while the `*ob_type` is a pointer to the *type object* for the given object. This type determines what the data represents, what kind of data it contains, and the kind of operations the object supports. Take the snippet in Listing 4.1 for example, the name, `name`, points to a string object, and the type of the object is “`str`”.

Listing 4.1: Variable declaration in python

```

1  >>> name = 'obi'
2  >>> type(name)
3  <class 'str'>

```

A valid question from here is *if the type field points to a type object then what does the `*ob_type` field of that type object point to?* The `ob_type` for a type object recursively refers to itself hence the saying that the type of a type is type.



A word on reference counting.

CPython uses reference counting for memory management. Reference counting is a simple method in which whenever a new reference to an object, such as binding a name to an object as in listing 4.1, is created the reference count of the object goes up by 1. The converse is true - whenever a reference to an object goes away (for example, using a `del` on `name` deletes the reference), the reference count is decremented by 1. When the reference count of an object hits zero, the VM can then deallocate that object. In the VM world, the `Py_INCREF` and `Py_DECREF` are used to increase and decrease the reference count of objects and they show up in a lot of code snippets that we discuss.

Types in the VM are implemented using the `_typeobject` data structure defined in the `Objects/Object.h` module. This is a C struct with fields for mostly functions or collections of functions filled in by each type. We look at this data structure next.

4.2 Dissecting Types

The `_typeobject` structure defined in `Include/Object.h` serves as the base structure of **all** Python types. The data structure defines a large number of fields that are mostly pointers to C functions that implement some functionality for a given type. Listing 4.2 is the `_typeobject` structure definition.

Listing 4.2: PyTypeObject definition

```
1 typedef struct _typeobject {
2     PyObject_VAR_HEAD
3     const char *tp_name; /* For printing, in format "<module>.<name>" */
4     Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */
5
6     destructor tp_dealloc;
7     printfunc tp_print;
8     getattrfunc tp_getattr;
9     setattrfunc tp_setattr;
10    PyAsyncMethods *tp_as_asyn;
11
12    reprfunc tp_repr;
13
14    PyNumberMethods *tp_as_number;
15    PySequenceMethods *tp_as_sequence;
16    PyMappingMethods *tp_as_mapping;
17
18    hashfunc tp_hash;
19    ternaryfunc tp_call;
20    reprfunc tp_str;
21    getattrofunc tp_getattro;
22    setattrofunc tp_setattro;
23
24    PyBufferProcs *tp_as_buffer;
25    unsigned long tp_flags;
26    const char *tp_doc; /* Documentation string */
27
28    traverseproc tp_traverse;
29
30    inquiry tp_clear;
31    richcmpfunc tp_richcompare;
32    Py_ssize_t tp_weaklistoffset;
33
34    getiterfunc tp_iter;
35    iternextfunc tp_iternext;
36
37    struct PyMethodDef *tp_methods;
38    struct PyMemberDef *tp_members;
39    struct PyGetSetDef *tp_getset;
40    struct _typeobject *tp_base;
41    PyObject *tp_dict;
42    descrgetfunc tp_descr_get;
```

```

43     descrsetfunc tp_descr_set;
44     Py_ssize_t tp_dictoffset;
45     initproc tp_init;
46     allocfunc tp_alloc;
47     newfunc tp_new;
48     freefunc tp_free;
49     inquiry tp_is_gc;
50     PyObject *tp_bases;
51     PyObject *tp_mro;
52     PyObject *tp_cache;
53     PyObject *tp_subclasses;
54     PyObject *tp_weaklist;
55     destructor tp_del;
56
57     unsigned int tp_version_tag;
58     destructor tp_finalize;
59 } PyTypeObject;

```

The `PyObject_VAR_HEAD` field is an extension of the `PyObject` field discussed in the previous section; this extension adds an `ob_size` field for objects that have the notion of length. The [Python C API documentation](#)¹ contains a description of each of the fields in this object structure. The critical thing to note is that the fields in the structure each implement a part of the type's behavior. Most of these fields are part of what we can call an object interface or protocol; the types implement these functions but in a type-specific way. For example, `tp_hash` field is a reference to a hash function for a given type - this field could be left without a value in the case where instances of the type are not hashable; whatever function is in the `tp_hash` field gets invoked when the `hash` method is called on an instance of that type. The type object also has the field - `tp_methods` that references methods unique to that type. The `tp_new` slot refers to a function that creates new instances of the type and so on. Some of these fields, such as `tp_init`, are optional - not every type needs to run an initialization function, especially when the type is immutable, such as tuples. In contrast, other fields, such as `tp_new`, are compulsory.

Also, among these fields are fields for other Python protocols, such as the following.

1. Number protocol - A type implementing this protocol will have implementations for the `PyNumberMethods *tp_as_number` field. This field is a reference to a set of functions that implement arithmetic operations (this does not necessarily have to be on numbers). A type will support arithmetic operations with their corresponding implementations included in the `tp_as_number` set in the type's specific way. For example, the non-numeric `set` type has an entry into this field because it supports arithmetic operations such as `-`, `<=`, and so on.

¹<https://docs.python.org/3.6/c-api/typeobj.html>

2. Sequence protocol - A type that implements this protocol will have a value in the `PySequenceMethods` `*tp_as_sequence` field. This value should provide that type with support for some [sequence operations](#)² such as `len`, `in` etc.
3. Mapping protocol - A type that implements this protocol will have a value in the `PyMappingMethods` `*tp_as_mapping`. This value enables such type to be treated like Python dictionaries using the dictionary syntax for setting and accessing key-value mappings.
4. Iterator protocol - A type that implements this protocol will have a value in the `getiterfunc` `tp_iter` and possibly the `iternextfunc` `tp_iternext` fields enabling instances of the type to be used like Python iterators.
5. Buffer protocol - A type implementing this protocol will have a value in the `PyBufferProcs` `*tp_as_buffer` field. These functions will enable access to the instances of the type as input/output buffers.

Next, we look at a few type objects as case studies of how the type object fields are populated.

4.3 Type Object Case Studies

The `tuple` type

We look at the `tuple` type to get a feel for how the fields of a type object are populated. We choose this because it is relatively easy to grok given the small size of the implementation - roughly a thousand plus lines of C including documentation strings. Listing 4.3 shows the implementation of the `tuple` type.

Listing 4.3: Tuple type definition

```

1 PyTypeObject PyTuple_Type = {
2     PyVarObject_HEAD_INIT(&PyType_Type, 0)
3     "tuple",
4     sizeof(PyTupleObject) - sizeof(PyObject *),
5     sizeof(PyObject *),
6     (destructor)tupledealloc,           /* tp_dealloc */
7     0,                                /* tp_print */
8     0,                                /* tp_getattr */
9     0,                                /* tp_setattr */
10    0,                                /* tp_reserved */
11    (reprfunc)tuplerepr,              /* tp_repr */
12    0,                                /* tp_as_number */
13    &tuple_as_sequence,                /* tp_as_sequence */
14    &tuple_as_mapping,                /* tp_as_mapping */
15    (hashfunc)tuplehash,              /* tp_hash */

```

²<https://docs.python.org/3.6/library/stdtypes.html#sequence-types-list-tuple-range>

```

16      0,                                     /* tp_call */
17      0,                                     /* tp_str */
18      PyObject_GenericGetAttr,                /* tp_getattro */
19      0,                                     /* tp_setattro */
20      0,                                     /* tp_as_buffer */
21      Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HAVE_GC |
22          Py_TPFLAGS_BASETYPE | Py_TPFLAGS_TUPLE_SUBCLASS, /* tp_flags */
23      tuple_doc,                            /* tp_doc */
24      (traverseproc)tupletraverse,           /* tp_traverse */
25      0,                                     /* tp_clear */
26      tuplerichcompare,                     /* tp_richcompare */
27      0,                                     /* tp_weaklistoffset */
28      tuple_iter,                           /* tp_iter */
29      0,                                     /* tp_iternext */
30      tuple_methods,                        /* tp_methods */
31      0,                                     /* tp_members */
32      0,                                     /* tp_getset */
33      0,                                     /* tp_base */
34      0,                                     /* tp_dict */
35      0,                                     /* tp_descr_get */
36      0,                                     /* tp_descr_set */
37      0,                                     /* tp_dictoffset */
38      0,                                     /* tp_init */
39      0,                                     /* tp_alloc */
40      tuple_new,                            /* tp_new */
41      PyObject_GC_Del,                      /* tp_free */
42  };

```

We look at the fields that are populated in this type.

1. PyObject_VAR_HEAD has been initialized with a type object - PyType_Type as the type. Recall that the type of a type object is Type. A look at the PyType_Type type object shows that the type of PyType_Type is itself.
2. tp_name is initialized to the name of the type - tuple.
3. tp_basicsize and tp_itemsize refer to the size of the tuple object and items contained in the tuple object, and this is filled in accordingly.
4. tupledealloc is a memory management function that handles the deallocation of memory when a tuple object is destroyed.
5. tuplerepr is the function invoked when the `repr` function is called with a tuple instance as an argument.
6. tuple_as_sequence is the set of sequence methods that the tuple implements. Recall that a tuple support `in`, `len` etc. sequence methods.

7. `tuple_as_mapping` is the set of mapping methods supported by a tuple - in this case, the keys are integer indexes only.
8. `tuplehash` is the function that is invoked whenever the hash of a tuple object is required. This comes into play when tuples are used as dictionary keys or in sets.
9. `PyObject_GenericGetAttr` is the generic function invoked when referencing attributes of a tuple object. We look at attribute referencing in subsequent sections.
10. `tuple_doc` is the documentation string for a tuple object.
11. `tupletraverse` is the traversal function for garbage collection of a tuple object. This function is used by the garbage collector to help in the detection of reference cycle³.
12. `tuple_iter` is a method that gets invoked when the `iter` function is called on a tuple object. In this case, a completely different `tuple_iterator` type is returned so there is no implementation for the `tp_iternext` method.
13. `tuple_methods` are the actual methods of a tuple type.
14. `tuple_new` is the function invoked to create new instances of tuple type.
15. `PyObject_GC_Del` is another field that references a memory management function.

The additional fields with \emptyset values are empty because tuples do not require those functionalities. Take the `tp_init` field, for example, a tuple is an immutable type, so once created it cannot be changed, so there is no need for any initialization beyond what happens in the function referenced by `tp_new`; hence this field is left empty.

The `type` type

The other type we look at is the `type` type. This is the *metatype* for all built-in types and vanilla user-defined types (a user can define a new metatype) - notice how this type is used when initializing the tuple object in `PyVarObject_HEAD_INIT`. When discussing types, it is essential to distinguish between objects that have `type` as their type and objects with a user-defined type. This distinction comes very much to the fore when dealing with attribute referencing in objects.

This type defines methods used when working with other types, and the fields are similar to those from the previous section. When creating new types, as we will see in subsequent sections, this type is used.

The `object` type

Another necessary type is the `object` type; this is very similar to the `type` type. The `object` type is the root type for all user-defined types and provides some default values that fill in the type fields of a user-defined type. This is because user-defined types behave differently compared to types that have `type` as their type. As we will see in subsequent section, functions such as that for the attribute resolution algorithm provided by the `object` type differ significantly from those offered by the `type` type.

³https://docs.python.org/3.6/c-api/typeobj.html#c.PyTypeObject.tp_traverse.

4.4 Minting type instances

With an assumed firm understanding of the rudiments of types, we can progress onto one of the most fundamental functions of types, which is the creation of new instances. To fully understand the process of creating new type instances, it is important to remember that just as we differentiate between built-in types and user-defined types ⁴, the internal structure of both differs. The `tp_new` field is the cookie cutter for new type instances in Python. The [documentation](#)⁵ for the `tp_new` slot as reproduced below gives a brilliant description of the function that should fill this slot.

An optional pointer to an instance creation function. If this function is NULL for a particular type, that type cannot be called to create new instances; presumably, there is some other way to create instances, like a factory function. The function signature is

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)
```

The `subtype` argument is the type of the object being created; the `args` and `kwds` arguments are the positional and keyword arguments of the call to the type. Note that `subtype` doesn't have to equal the type whose `tp_new` function is called; it may be a subtype of that type (but not an unrelated type). The `tp_new` function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the `tp_init` handler. A good rule of thumb is that for immutable types, all initialization should take place in `tp_new`, while for mutable types, most initialization should be deferred to `tp_init`.

This field is inherited by subtypes but not by static types whose `tp_base` is NULL or `&PyBaseObject_Type`.

We will use the `tuple` type from the previous section as an example of a builtin type. The `tp_new` field of the `tuple` type references the `-tuple_new` method shown in Listing 4.4, which handles the creation of new tuple objects. A new tuple object is created by dereferencing and then invoking this function.

⁴Type is used rather than class for uniformity

⁵https://docs.python.org/3/c-api/typeobj.html#c.PyTypeObject.tp_new

Listing 4.4: tuple_new function for creating new tuple instances

```

1 static PyObject * tuple_new(PyTypeObject *type, PyObject *args,
2                             PyObject *kwds){
3     PyObject *arg = NULL;
4     static char *kwlist[] = {"sequence", 0};
5
6     if (type != &PyTuple_Type)
7         return tuple_subtype_new(type, args, kwds);
8     if (!PyArg_ParseTupleAndKeywords(args, kwds, "|O:tuple", kwlist, &arg))
9         return NULL;
10
11    if (arg == NULL)
12        return PyTuple_New(0);
13    else
14        return PySequence_Tuple(arg);
15 }
```

Ignoring the first and second conditions for creating a tuple in Listing 4.4, we follow the third condition, `if (arg==NULL) return PyTuple_New(0)` down the rabbit hole to find out how this works. Overlooking the optimizations abound in the `PyTuple_New` function, the segment of the function that creates a new tuple object is the `op = PyObject_GC_NewVar(PyTupleObject, &PyTuple_Type, size)` call which allocates memory for an instance of the `PyTuple_Obje`ct structure on the heap. This is where a difference between internal representation of built-in types and user-defined types comes to the fore - instances of built-ins like `tuple` are actually C structures. So what does this C struct backing a tuple object look like? It is found in the `Include/tupleobject.h` as the `PyTupleObject` typedef, and this is shown in Listing 4.5 for convenience.

Listing 4.5: PyTuple_Obje

```

1 typedef struct {
2     PyObject_VAR_HEAD
3     PyObject *ob_item[1];
4
5     /* ob_item contains space for 'ob_size' elements.
6      * Items must typically not be NULL, except during construction when
7      * the tuple is not yet visible outside the function that builds it.
8      */
9 } PyTupleObject;
```

The `PyTupleObject` is defined as a struct having a `PyObject_VAR_HEAD` and an array of `PyObject` pointers - `ob_items`. This leads to a very efficient implementation as opposed to representing the instance using Python data structures.

Recall that an object is a collection of methods and data. The `PyTupleObject` in this case provides space to hold the actual data that each tuple object contains so we can have multiple instances of `PyTupleObject` allocated on the heap but these will all reference the single `PyTuple_Type` type that provides the methods that can operate on this data.

Now consider a user-defined class such as in Listing 4.6.

Listing 4.6: User defined class

```
1 class Test:
2     pass
```

The `Test` type, as you would expect, is an object of `instance_Type`. To create an instance of the `Test` type, the `Test` type is called as such - `Test()`. As always, we can go down the rabbit hole to convince ourselves of what happens when the type object is called. The `Type` type has a function reference - `type_call` that fills the `tp_call` field, and this is dereferenced whenever the *call* notation is used on an instance of `Type`. A snippet of the `type_call` the function implementation is shown in listing 4.7.

Listing 4.7: A snippet of `type_call` function definition

```
1 ...
2     obj = type->tp_new(type, args, kwds);
3     obj = _Py_CheckFunctionResult((PyObject*)type, obj, NULL);
4     if (obj == NULL)
5         return NULL;
6
7     /* Ugly exception: when the call was type(something),
8      don't call tp_init on the result. */
9     if (type == &PyType_Type &&
10         PyTuple_Check(args) && PyTuple_GET_SIZE(args) == 1 &&
11         (kwds == NULL || (PyDict_Check(kwds) && PyDict_Size(kwds) == 0)))
12         return obj;
13
14     /* If the returned object is not an instance of type,
15      it won't be initialized. */
16     if (!PyType_IsSubtype(Py_TYPE(obj), type))
17         return obj;
18
19     type = Py_TYPE(obj);
20     if (type->tp_init != NULL) {
21         int res = type->tp_init(obj, args, kwds);
22         if (res < 0) {
23             assert(PyErr_Occurred());
24             Py_DECREF(obj);
25         }
26     }
27 }
```

```

26         obj = NULL;
27     }
28     else {
29         assert(!PyErr_Occurred());
30     }
31 }
32 return obj;

```

In Listing 4.7, we see that when a Type object instance is called, the function referenced by the `tp_new` field is invoked to create a new instance of that type. The `tp_init` function is also called if it exists to initialize the new instance. This process explains builtin types because, after all, they have their own `tp_new` and `tp_init` functions defined already, but what about user-defined types? Most times, a user does not define a `__new__` function for a new type (when defined, this will go into the `tp_new` field during class creation). The answer to this also lies with the `type_new` function that fills the `tp_new` field of the Type. When creating a user-defined type, such as `Test`, the `type_new` function checks for the presence of base types (supertypes/classes) and when there are none, the `PyBaseObject_Type` type is added as a default base type, as shown in listing 4.8.

Listing 4.8: Snippet showing how the `PyBaseObject_Type` is added to list of bases

```

...
if (nbases == 0) {
    bases = PyTuple_Pack(1, &PyBaseObject_Type);
    if (bases == NULL)
        goto error;
    nbases = 1;
}
...

```

This default base type defined in the `Objects/typeobject.c` module contains some default values for the various fields. Among these defaults are values for the `tp_new` and `tp_init` fields. These are the values that get called by the interpreter for a user-defined type. In the case where the user-defined type implements its methods such as `__init__`, `__new__` etc., these values are called rather than those of the `PyBaseObject_Type` type.

One may notice that we have not mentioned any object structures like the tuple object structure, `tupleobject`, and ask - *if no object structures are defined for a user-defined class, how are object instances handled and where do objects attributes that do not map to slots in the type reside?* This has to do with the `tp_dictoffset` field - a numeric field in type object. Instances are created as `PyObjects`, however, when this offset value is non-zero in the instance type, it specifies the offset of the instance attribute dictionary from the instance (the `PyObject`) itself as shown in figure 4.0 so for an instance of a `Person` type, the attribute dictionary can be assessed by adding this offset value to the origin of the `PyObject` memory location.

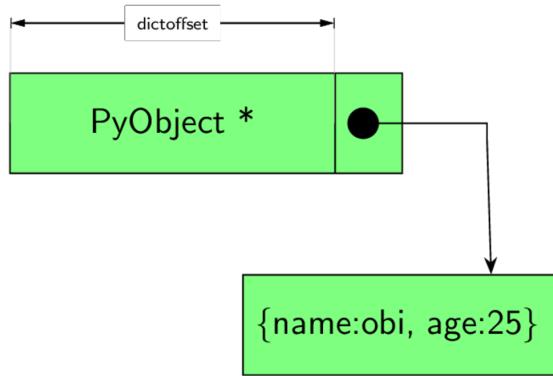


Figure 4.0: How instances of user-defined types are structured.

For example, if an instance *PyObject* is at 0x10 and the offset is 16 then the instance dictionary that contains instance attributes is found at 0x10 + 16. This is not the only way instances store their attributes, as we will see in the following section.

4.5 Objects and their attributes

Types and their attributes (*variables and methods*) are central to object-oriented programming. Conventionally, types and instances store their attributes using a `dict` data structure, but this is not the full story in cases of instances that have the `__slots__` attribute defined. This `dict` data structure resides in one of two places, depending on the type of the object, as was mentioned in the previous section.

1. For objects with a type of `Type`, the `tp_dict` slot of type structure is a pointer to a `dict` that contains values, variables, and methods for that type. In the more conventional sense, we say the `tp_dict` field of the type object data structure is a pointer to the type or `class` `dict`.
2. For objects with user-defined types, that `dict` data structure when present is located just after the `PyObject` structure that represents the object. The `tp_dictoffset` value of the type of the object gives the offset from the start of an object to this instance `dict` contains the instance attributes.

Performing a simple dictionary access to obtain attributes looks simpler than it actually is. Infact, searching for attributes is way more involved than just checking `tp_dict` value for instance of `Type` or the `dict` at `tp_dictoffset` for instances of user-defined types. To get a full understanding, we have to discuss the *descriptor protocol* - a protocol at the heart of attribute referencing in Python.

The [Descriptor HowTo Guide](https://docs.python.org/3/howto/descriptor.html)⁶ is an excellent introduction to descriptors, but the following section provides a cursory description of descriptors. Simply put, a *descriptor* is an **object** that implements the `__get__`, `__set__` or `__delete__` special methods of the descriptor protocol. Listing 4.9 is the signature for each of these methods in Python.

⁶<https://docs.python.org/3/howto/descriptor.html>

Listing 4.9: The Descriptor protocol methods

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Objects implementing only the `__get__` method are non-data descriptors so they can only be read from after initialization. In contrast, objects implementing the `__get__` and `__set__` are data descriptors meaning that such descriptor objects are writeable. We are interested in the application of descriptors to object attribute representation. The `TypedAttribute` descriptor in listing 4.10 is an example of a descriptor used to represent an object attribute.

Listing 4.10: A simple descriptor for type checking attribute values

```
class TypedAttribute:

    def __init__(self, name, type, default=None):
        self.name = "_" + name
        self.type = type
        self.default = default if default else type()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        raise AttributeError("Can't delete attribute")
```

The `TypedAttribute` descriptor class enforces rudimentary type checking for any class' attribute that it represents. It is important to note that descriptors are useful in this kind of case only when defined at the class level rather than instance-level, i.e., in `__init__` method, as shown in listing 4.11.

Listing 4.11: Type checking on instance attributes using TypedAttribute descriptor

```

class Account:
    name = TypedAttribute("name", str)
    balance = TypedAttribute("balance", int, 42)

    def name_balance_str(self):
        return str(self.name) + str(self.balance)

>> acct = Account()
>> acct.name = "obi"
>> acct.balance = 1234
>> acct.balance
1234
>> acct.name
obi
# trying to assign a string to number fails
>> acct.balance = '1234'
TypeError: Must be a <type 'int'>

```

If one thinks carefully about it, it only makes sense for this kind of descriptor to be defined at the type level because if defined at the instance level, then any assignment to the attribute will overwrite the descriptor.

A review of the Python VM source code shows the importance of descriptors. Descriptors provide the mechanism behind properties, static methods, class methods, and a host of other functionality in Python. Listing 4.12, the algorithm for resolving an attribute from an instance, *b*, of a user-defined type, is a concrete illustration of the importance of descriptors.

Listing 4.12: Algorithm to find a referenced attribute in an instance of a user-defined type

-
1. Search `type(b).__dict__` for the attribute by name. If the attribute is present and a data descriptor, then return the result of calling the descriptor's `__get__` method. If the name is absent, then all base classes in the *MRO* of `type(b)` are searched in the same way.
 2. Search `b.__dict__`, and if the attribute is present, return it.
 3. if the name from 1 is a non-data descriptor return the value of calling `__get__`,
 4. If the name is not found, call `__getattr__()` if provided by the user-defined type otherwise raise an `AttributeError`.
-

The algorithm in Listing 4.12 shows that during attribute referencing we first check for descriptor objects; it also illustrates how the `TypedAttribute` descriptor can represent an attribute of an object - whenever an attribute is referenced such as `b.name` the VM searches the `Account` type object for the

attribute, and in this case, it finds a `TypedAttribute` descriptor; the VM then calls `__get__` method of the descriptor accordingly. The `TypedAttribute` example illustrates a descriptor but is somewhat contrived; to get a real touch of how important descriptors are to the core of the language, we consider some examples that show how they are applied.

Do note that the attribute reference algorithm in listing 4.12 differs from the algorithm used when referencing an attribute whose type is `type`. Listing 4.3 shows the algorithm for such.

Listing 4.13: Algorithm to find a referenced attribute in a type

1. Search `type(type).__dict__` for the attribute by name. If the name is present and it is a data descriptor, return the result of calling the descriptor's `__get__` method. If the name is absent, then all base classes in the *MRO* of `type(type)` are searched in the same way.
 2. Search `type.__dict__` and all its bases for the attribute by name. If the name present and it is a descriptor, then invoke its `__get__` method, otherwise return the value.
 3. If a value was found in (1) and it is a non-data descriptor, then return the value from invoking its `__get__` function.
 4. if the value found in (1) is not a descriptor, then return it.
-

Examples of Attribute Referencing with Descriptors inside the VM

Consider the type data structure discussed earlier in this chapter. The `tp_descr_get` and `tp_descr_set` fields in a type data structure can be filled in by any type instance to satisfy the descriptor protocol. A function object is a perfect place to show how this works.

Given the type definition, `Account` from listing 4.11, consider what happens when we reference the method, `name_balance_str`, from the class as such - `Account.name_balance_str` and when we reference the same method from an instance as shown in listing 4.14.

Listing 4.14: Illustrating bound and unbound functions

```
>> a = Account()
>> a.name_balance_str
<bound method Account.name_balance_str of <__main__.Account object at
0x102a0ae10>>

>> Account.name_balance_str
<function Account.name_balance_str at 0x102a2b840>
```

Looking at the snippet from listing 4.14, although we seem to reference the same attribute, the actual objects returned are different in value and type. When referenced from the `Account` type, the returned value is a `function` type, but when referenced from an instance of the `Account` type, the result is a `bound method` type. This is possible because functions are descriptors too. Listing 4.15 is the definition of a `function` object type.

Listing 4.15: Function type object definition

```
PyTypeObject PyFunction_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "function",
    sizeof(PyFunctionObject),
    0,
    (destructor)func_dealloc, /* tp_dealloc */
    0, /* tp_print */
    0, /* tp_getattr */
    0, /* tp_setattr */
    0, /* tp_reserved */
    (reprfunc)func_repr, /* tp_repr */
    0, /* tp_as_number */
    0, /* tp_as_sequence */
    0, /* tp_as_mapping */
    0, /* tp_hash */
    function_call, /* tp_call */
    0, /* tp_str */
    0, /* tp_getattro */
    0, /* tp_setattro */
    0, /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HAVE_GC, /* tp_flags */
    func_doc, /* tp_doc */
    (traverseproc)func_traverse, /* tp_traverse */
    0, /* tp_clear */
    0, /* tp_richcompare */
    offsetof(PyFunctionObject, func_weakreflist), /* tp_weaklistoffset */
    0, /* tp_iter */
    0, /* tp_iternext */
    0, /* tp_methods */
    func_memberlist, /* tp_members */
    func_getsetlist, /* tp_getset */
    0, /* tp_base */
    0, /* tp_dict */
    func_descr_get, /* tp_descr_get */
    0, /* tp_descr_set */
    offsetof(PyFunctionObject, func_dict), /* tp_dictoffset */
    0, /* tp_init */
    0, /* tp_alloc */
    func_new, /* tp_new */
};
```

The function object fills in the `tp_descr_get` field with a `func_descr_get` function thus instances of the `function` type are non-data descriptors. Listing 4.16 shows the implementation of the `func_descr_get` method.

Listing 4.16: Function type object definition

```
static PyObject * func_descr_get(PyObject *func, PyObject *obj, PyObject *type){
    if (obj == Py_None || obj == NULL) {
        Py_INCREF(func);
        return func;
    }
    return PyMethod_New(func, obj);
}
```

The `func_descr_get` can be invoked during either type attribute resolution or instance attribute resolution, as described in the previous section. When invoked from a type, the call to the `func_descr_get` is as such `local_get(attribute, (PyObject *)NULL, (PyObject *)type)` while when invoked from an attribute reference of an instance of a user-defined type, the call signature is `f(descr, obj, (PyObject *)Py_TYPE(obj))`. Going over the implementation for `func_descr_get` in listing 4.16, we see that if the instance is `NULL`, then the function itself is returned while if an instance is passed in to the call, a new method object is created using the function and the instance. This sums up how Python can return a different type for the same function reference using a descriptor.



The `self` argument is the first parameter to any instance method definition, but why is it left out when such methods are called? The reality is that instance methods do take the instance (called `self` by convention) as the first argument - this is just transparent to the caller. A call such as `b.name_balance_str()` is the same thing as `type(b).name_balance_str(b)`. The reason we can invoke `b.name_balance_str()` is that the value returned by `b.name_balance_str` is a method object which is a thin wrapper around the `name_balance_str` with the instance already bound to the method instance. So when we make a call such as `b.name_balance_str()` the method uses the bound instance as an argument to the wrapped function hiding this detail from us.

In another instance of the importance of descriptors, consider the snippet in Listing 4.17 which shows the result of accessing the `__dict__` attribute from both an instance of the built-in type and an instance of a user-defined type.

Listing 4.17: Accessing the `__dict__` attribute from an instance of the built-in type and an instance of a user-defined type

```

class A:
    pass

>>> A.__dict__
mappingproxy({('__module__': '__main__', '__doc__': None, '__weakref__': <attribute '__weakref__' of 'A' objects>, '__dict__': <attribute '__dict__' of 'A' objects>})
>>> i = A()
>>> i.__dict__
{}
>>> A.__dict__['name'] = 1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> i.__dict__['name'] = 2
>>> i.__dict__
{'name': 2}
>>>

```

Observe from listing 4.17 that both objects do not return the vanilla dictionary type when the `__dict__` attribute is referenced. The type object seems to return an immutable mapping proxy that we cannot even assign. In contrast, the instance of type returns a vanilla dictionary mapping that supports all the usual dictionary functions. So it seems that attribute referencing is done differently for these objects. Recall the algorithm described for attribute search from a couple of sections back. The first step is to search the `__dict__` of the type of the object for the attribute, so we go ahead and do this for both objects in listing 4.18.

Listing 4.18: Checking for `__dict__` in type of objects

```

>>> type(type.__dict__['__dict__']) # type of A is type
<class 'getset_descriptor'>
    type(A.__dict__['__dict__'])
<class 'getset_descriptor'>

```

We see that the `__dict__` attribute is represented by data descriptors for both objects, and that is why we can get different object types. We would like to find out what happens under the covers for this descriptor, just as we did in the *functions* and *bound methods*. A good place to start is the `Objects/typeobject.c` module and the definition for the `type` type object. An interesting field is the `tp_getset` field that contains an array of C structs (`PyGetSetDef` values) shown in listing 4.19. This is the collection of values that will be represented by descriptors in `type`'s type `__dict__` attribute - the `__dict__` attribute is the mapping referred to by the `tp_dict` slot of the `type` object points.

Listing 4.19: Checking for `__dict__` in type of objects

```

static PyGetSetDef type_getsets[] = {
    {"__name__", (getter)type_name, (setter)type_set_name, NULL},
    {"__qualname__", (getter)type_qualname, (setter)type_set_qualname, NULL},
    {"__bases__", (getter)type_get_bases, (setter)type_set_bases, NULL},
    {"__module__", (getter)type_module, (setter)type_set_module, NULL},
    {"__abstractmethods__", (getter)type_abstractmethods,
        (setter)type_set_abstractmethods, NULL},
    {"__dict__", (getter)type_dict, NULL, NULL},
    {"__doc__", (getter)type_get_doc, (setter)type_set_doc, NULL},
    {"__text_signature__", (getter)type_get_text_signature, NULL, NULL},
    {0}
};

```

These values are not the only ones represented by descriptors in the type *dict*; there are other values such as the `tp_members` and `tp_methods` values which have descriptors created and inserted into the `tp_dict` during type initialization. The insertion of these values into the *dict* happens when the `PyType_Ready` function is called on the type. As part of the `PyType_Ready` function initialization process, descriptor objects are created for each entry in the `type_getsets` and then added into the `tp_dict` mapping - the `add_getset` function in the `Objects/typeobject.c` handles this.

Returning to our `__dict__` attribute, we know that after initialization of the type, the `__dict__`-attribute exists in the `tp_dict` field of the type, so let's see what the `getter` function of this descriptor does. The `getter` function is the `type_dict` function shown in listing 4.20.

Listing 4.20: Getter function for an instance of `type`

```

static PyObject * type_dict(PyTypeObject *type, void *context){
    if (type->tp_dict == NULL) {
        Py_INCREF(Py_None);
        return Py_None;
    }
    return PyDictProxy_New(type->tp_dict);
}

```

The `tp_getattro` field points to the function that is the first port of call for getting attributes for any object. For the type object, it points to the `type_getattro` function. This method, in turn, implements the attribute search algorithm as described in listing 4.13. The function invoked by the descriptor found in the type dict for the `__dict__` attribute is the `type_dict` function given in listing 4.20, and it is pretty easy to understand. The return value is of interest to us here; it is a dictionary proxy to the actual dictionary that holds the `type` attribute; this explains the `mappingproxy` type that is returned when we query the `__dict__` attribute of a type object.

So what about the instance of A, a user-defined type, how is the `__dict__` attribute resolved? Now recall that A is an object of type `type` so we go hunting in the `Object/typeobject.c` module to see how new type instances are created. The `tp_new` slot of the `PyType_Type` contains the `type_new` function that handles the creation of new type objects. Perusing through all the type creation code in the function, one stumbles on the snippet in listing 4.21.

Listing 4.21: Setting `tp_getset` field for user defined type

```
if (type->tp_weaklistoffset && type->tp_dictoffset)
    type->tp_getset = subtype_getsets_full;
else if (type->tp_weaklistoffset && !type->tp_dictoffset)
    type->tp_getset = subtype_getsets_weakref_only;
else if (!type->tp_weaklistoffset && type->tp_dictoffset)
    type->tp_getset = subtype_getsets_dict_only;
else
    type->tp_getset = NULL;
```

Assuming the first conditional is true as the `tp_getset` field is filled with the value shown in Listing 4.22.

Listing 4.22: The `getset` values for instance of `type`

```
static PyGetSetDef subtype_getsets_full[] = {
    {"__dict__", subtype_dict, subtype_setdict,
     PyDoc_STR("dictionary for instance variables (if defined)")},
    {"__weakref__", subtype_getweakref, NULL,
     PyDoc_STR("list of weak references to the object (if defined)")},
    {0}
};
```

When `(*tp->tp_getattro)(v, name)` is invoked, the `tp_getattro` field which contains a pointer to the `PyObject_GenericGetAttr` is called. This function is responsible for implementing the attribute search algorithm for a user-defined types. In the case of the `__dict__` attribute, the descriptor is found in the object type's `dict` and the `__get__` function of the descriptor is the `subtype_dict` function defined for the `__dict__` attribute from listing 4.21. The `subtype_dict` *getter* function is shown in listing 4.23.

Listing 4.23: The *getter* function for `__dict__` attribute of a user-defined type

```

static PyObject * subtype_dict(PyObject *obj, void *context){
    PyTypeObject *base;

    base = get_builtin_base_with_dict(Py_TYPE(obj));
    if (base != NULL) {
        descrgetfunc func;
        PyObject *descr = get_dict_descriptor(base);
        if (descr == NULL) {
            raise_dict_descr_error(obj);
            return NULL;
        }
        func = Py_TYPE(descr)->tp_descr_get;
        if (func == NULL) {
            raise_dict_descr_error(obj);
            return NULL;
        }
        return func(descr, obj, (PyObject *)(Py_TYPE(obj)));
    }
    return PyObject_GenericGetDict(obj, context);
}

```

The `get_builtin_base_with_dict` returns a value when the object instance is in an inheritance hierarchy, so ignoring that for this instance is appropriate. The `PyObject_GenericGetDict` object is invoked. Listing 4.24 shows the `PyObject_GenericGetDict` and an associated helper that fetches the instance dict. The actual *get the dict* function is the `_PyObject_GetDictPtr` function that queries the object for its `dictoffset` and uses that to compute the address of the instance dict. In a situation where this function returns a null value, `PyObject_GenericGetDict` can return a new dict to the calling function.

Listing 4.24: Fetching dict attribute of an instance of a user defined type

```

PyObject * PyObject_GenericGetDict(PyObject *obj, void *context){
    PyObject *dict, **dictptr = _PyObject_GetDictPtr(obj);
    if (dictptr == NULL) {
        PyErr_SetString(PyExc_AttributeError,
                       "This object has no __dict__");
        return NULL;
    }
    dict = *dictptr;
    if (dict == NULL) {
        PyTypeObject *tp = Py_TYPE(obj);
        if ((tp->tp_flags & Py_TPFLAGS_HEAPTYPE) && CACHED_KEYS(tp)) {

```

```

        DK_INCREF(CACHED_KEYS(tp));
        *dictptr = dict = new_dict_with_shared_keys(CACHED_KEYS(tp));
    }
    else {
        *dictptr = dict = PyDict_New();
    }
}
Py_XINCREF(dict);
return dict;
}

PyObject ** _PyObject_GetDictPtr(PyObject *obj){
    Py_ssize_t dictoffset;
    PyTypeObject *tp = Py_TYPE(obj);

    dictoffset = tp->tp_dictoffset;
    if (dictoffset == 0)
        return NULL;
    if (dictoffset < 0) {
        Py_ssize_t tsize;
        size_t size;

        tsize = ((PyVarObject *)obj)->ob_size;
        if (tsize < 0)
            tsize = -tsize;
        size = _PyObject_VAR_SIZE(tp, tsize);

        dictoffset += (long)size;
        assert(dictoffset > 0);
        assert(dictoffset % SIZEOF_VOID_P == 0);
    }
    return (PyObject **) ((char *)obj + dictoffset);
}

```

This explanation succinctly sums up how the Python VM uses descriptors to implement type-dependent custom attribute access depending on types. Descriptors are pervasive in the VM; `__slots__`, static and class methods, properties are just some further examples of language features that are made possible by the use of descriptors.

4.6 Method Resolution Order (MRO)

We have mentioned *MRO* when discussing attribute referencing without discussing it much so in this section, we go into a bit more detail on *MRO*. Types can belong to a multiple inheritance hierarchy, so there is a need for some kind of order defining how to search for methods when a type inherits from multiple classes; this order which is referred to as *Method Resolution Order (MRO)* is also actually used when searching for other non-method attributes as we saw in the algorithm for attribute reference resolution. The article, [Python 2.3 Method Resolution order⁷](#), is an excellent and easy to read documentation of the method resolution algorithm used in Python; a summary of the main points are reproduced here.

Python uses the [C3⁸](#) algorithm for building the *method resolution order* (also referred to as *linearization* here) when a type inherits from multiple base types. Listing 4.25 shows some notations used in explaining this algorithm.

`C1 C2 ... CN` denotes the list of classes `[C1, C2, C3 ..., CN]`

The head of the list is its first element: `head = C1`

The tail is the rest of the list: `tail = C2 ... CN`.

`C + (C1 C2 ... CN) = C C1 C2 ... CN` denotes the sum of the lists `[C] + [C1, C2, ..., CN]`.

Consider a type `C` in a multiple inheritance hierarchy, with `C` inheriting from the base types `B1, B2, ..., BN`, the linearization of `C` is the sum of `C` plus the merge of the linearizations of the parents and the list of the parents - `L[C(B1 ... BN)] = C + merge(L[B1] ... L[BN], B1 ... BN)`. The linearization of the object type which has no parents is trivial - `L[object] = object`. The `merge` operation is calculated according to the following [algorithm⁹](#):

take the head of the first list, i.e., `L[B1][0]`; if this head is not in the tail of any of the other lists, then add it to the linearization of `C` and remove it from the lists in the merge, otherwise look at the head of the next list and take it, if it is a good head. Then repeat the operation until all the classes have been removed, or it is impossible to find good heads. In this case, it is impossible to construct the merge; Python 2.3 will refuse to create the class `C` and will raise an exception.

Some type hierarchies cannot be linearized using this algorithm, and in such cases, the VM throws an error and does not create such hierarchies.

⁷<https://www.python.org/download/releases/2.3/mro/>

⁸<http://citeseerv.ist.psu.edu/viewdoc/download?doi=10.1.1.19.3910&rep=rep1&type=pdf>

⁹<https://www.python.org/download/releases/2.3/mro/>

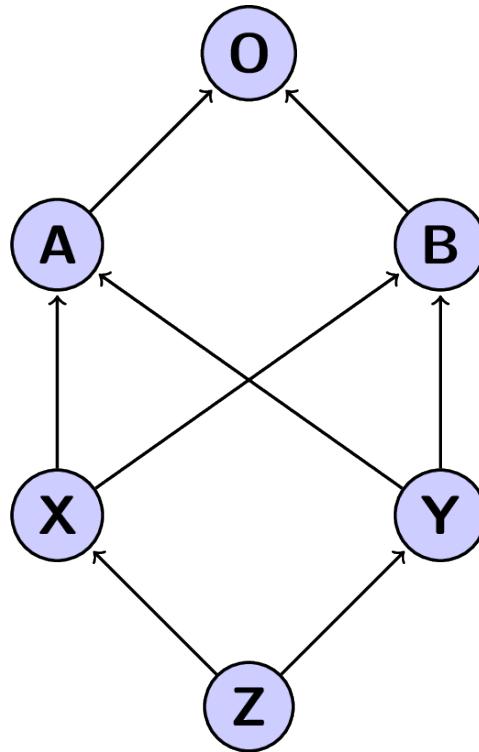


Figure 4.1: A simple multiple inheritance hierarchy.

Assuming we have an inheritance hierarchy such as that shown in figure 4.1, the algorithm for creating the *MRO* would proceed as follows starting from the top of the hierarchy with O, A, and B. The linearizations of O, A and B are trivial:

Listing 4.26: Calculating linearization for types O, A and B from figure 4.1

```

L[O] = O
L[A] = A O
L[B] = B O
  
```

The linearization of X can be computed as $L[X] = X + \text{merge}(AO, BO, AB)$

A is a good head, so it is added to the linearization, and we are left to compute $\text{merge}(AO, BO, AB)$. O is not a good head because it is in the tail of BO, so we move to the next sequence. B is a good head, so we add it to the linearization, and we are left to compute $\text{merge}(O, O)$, which evaluates to O. The resulting linearization of X - $L[X] = X A B O$.

Like the procedure from above, the linearization for Y is computed, as shown in Listing 4.27:

Listing 4.27: Calculating linearization for type Y from figure 4.1

```
L[Y] = Y + merge(A0, B0, AB)
      = Y + A + merge(0, B0, B)
      = Y + A + B + merge(0, 0)
      = Y A B 0
```

With linearizations for X and Y computed, we can compute that for Z as shown in listing 4.28.

Listing 4.28: Calculating linearization for type z from figure 4.1

```
L[Z] = Z + merge(XAB0, YAB0, XY)
      = Z + X + merge(AB0, YAB0, Y)
      = Z + X + Y + merge(AB0, ABO)
      = Z + X + Y + A + merge(B0, B0)
      = Z + X + Y + A + B + merge(0, 0)
      = Z X Y A B 0
```

5. Code Objects

Code objects are essential building blocks of the Python virtual machine. Code objects encapsulate the Python virtual machine's bytecode; we may call the bytecode the assembly language of the Python virtual machine.

Code objects, as the name suggests, represent compiled executable Python code. We had come across code objects before when we discussed Python source compilation. The compilation process maps each code block to a code object. As described in the brilliant [Python documentation](#)¹:

A Python program is constructed from code blocks. A block is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified as a command-line argument to the interpreter) is a code block. A script command (a command specified on the interpreter command line with the '-c' option) is a code block. The string argument passed to the built-in functions `eval()` and `exec()` is a code block.

The code object contains runnable bytecode instructions that alter the state of the Python VM when run. Given a function, we can access its code object using the `__code__` attribute as in the following snippet.

Listing 5.1: Function code objects

```
def return_author_name():
    return "obi Ike-Nwosu"

>>> return_author_name.__code__
<code object return_author_name at 0x102279270, file "<stdin>", line 1>
```

For other code blocks, one can obtain the code objects for that code block by compiling such code. The `compile`² function provides a facility for this in the Python interpreter. The code objects possess several fields that are used by the interpreter loop when executing and we look at some of these in the following sections.

5.1 Exploring code objects

An excellent way to start with code objects is to compile a simple function and inspect the resulting code object. We use the simple `fizzbuzz` function shown in Listing 5.2 as a guinea pig.

¹<https://docs.python.org/3/reference/executionmodel.html>

²<https://docs.python.org/3.6/library/functions.html#compile>

Listing 5.2: Function code objects attributes of Fizzbuzz function

```

co_argcount = 1
co_cellvars = ()
co_code = b'\x00d\x01\x16\x00d\x02k\x02r\x1e|\x00d\x03\x16\x00d\x02k\x02r\x1ed\\
x04S\x00n,|\x00d\x01\x16\x00d\x02k\x02r0d\x05S\x00n\x1a|\x00d\x03\x16\x00d\x02k\x02r\
Bd\x06S\x00n\x08t\x00|\x00\x83\x01S\x00d\x00S\x00'
co_consts = (None, 3, 0, 5, 'FizzBuzz', 'Fizz', 'Buzz')
co_filename = /Users/c4obi/projects/python_source/cpython/fizzbuzz.py
co_firstlineno = 6
co_flags = 67
co_freevars = ()
co_kwonlyargcount = 0
co_lnotab = b'\x00\x01\x18\x01\x06\x01\x0c\x01\x06\x01\x0c\x01\x06\x02'
co_name = fizzbuzz
co_names = ('str',)
co_nlocals = 1
co_stacksize = 2
co_varnames = ('n',)

```

The fields shown are almost self-explanatory except for the `co_lnotab` and `co_code` fields that seem to contain gibberish.

1. `co_argcount`: This is the number of arguments to a code block and has a value only for function code blocks. The value is set to the count of the argument set of the code block's AST during the compilation process. The evaluation loop makes use of these variables during the set-up for code evaluation to carry out sanity checks such as checks that all arguments are present and for storing locals.
2. `co_code`: This holds the sequence of bytecode instructions executed by the evaluation loop. Each of these bytecode instruction sequences is composed of an opcode and an oparg - arguments to the opcode where it exists. For example, `co.co_code[0]` returns the first byte of the instruction, 124 that maps to a Python `LOAD_FAST` opcode.
3. `co_consts`: This field is a list of constants like string literals and numeric values contained within the code object. The example from above shows the content of this field for the `fizzbuzz` function. The values included in this list are integral to code execution as they are the values referenced by the `LOAD_CONST` opcode. The operand argument to a bytecode instruction such as the `LOAD_CONST` is the index into this list of constants. Consider the `co_consts` value of `(None, 3, 0, 5, 'FizzBuzz', 'Fizz', 'Buzz')` for the `FizzBuzz` function and contrast with the disassembled code object below.

Listing 5.3: Cross section of bytecode instructions for Fizzbuzz function

0 LOAD_BUILD_CLASS	
2 LOAD_CONST	0 (<code object Test at 0x101a02810, file "fiz\\
zbuzz.py", line 1>)	
4 LOAD_CONST	1 ('Test')
6 MAKE_FUNCTION	0
8 LOAD_CONST	1 ('Test')
10 CALL_FUNCTION	2
12 STORE_NAME	0 (Test)
...	
66 LOAD_GLOBAL	0 (str)
68 LOAD_FAST	0 (n)
70 CALL_FUNCTION	1
72 RETURN_VALUE	
74 LOAD_CONST	0 (None)
76 RETURN_VALUE	

Recall that during the compilation process, a `return None` is added if there is no `return` statement at the end of a function so we can tell that the bytecode instruction at offset 74 is a `LOAD_CONST` for a `None` value. The opcode argument is a 0, and we can see that the `None` value has an index of 0 in the constants list from where the `LOAD_CONST` instruction loads it.

4. `co_filename`: This field, as the name suggests, contains the name of the file that contains the code object's source code from which the code object.
5. `co_firstlineno`: This gives the line number on which the source for the code object begins. This value plays quite an essential role during activities such as debugging code.
6. `co_flags`: This field indicates the kind of code object. For example, if the code object is that of a coroutine, the flag is set to `0x0080`. Other flags such as `CO_NESTED` indicate if a code object is nested within another code block, `CO_VARARGS` indicates if a code block has variable arguments. These flags affect the behaviour of the evaluation loop during bytecode execution.
7. `co_lnotab`: The contains a string of bytes used to compute the source line numbers that correspond to instruction at a bytecode offset. For example, the `dis` the function makes use of this when calculating line numbers for instructions.
8. `co_varnames`: This is the number of locally defined names in a code block. Contrast this with `co_names`.
9. `co_names`: This a collection of non-local names used within the code object. For example, the snippet in listing 5.4 references a non-local variable, `p`.

Listing 5.4: Illustrating local and non-local names

```
def test_non_local():
    x = p + 1
    return x
```

List 5.5 is the result of introspecting on the code object for the function in Listing 5.4.

Listing 5.5: Illustrating local and non-local names

```
co_argcount = 0
co_cellvars = ()
co_code = b't\x00d\x01\x17\x00}\x00|\x00S\x00'
co_consts = (None, 1)
co_filename = /Users/c4obi/projects/python_source/cpython/fizzbuzz.py
co_firstlineno = 18
co_flags = 67
co_freevars = ()
co_kwonlyargcount = 0
co_lnotab = b'\x00\x01\x08\x01'
co_name = test_non_local
co_names = ('p',)
co_nlocals = 1
co_stacksize = 2
co_varnames = ('x',)
```

From this example, the difference between the `co_names` and `co_varnames` is noticeable. `co_varnames` references the locally defined names while `co_names` references non-locally defined names. Do note that it is only during execution of the program that an error is raised when the name `p` is not found. Listing 5.6 shows the bytecode instructions for the function in Listing 5.4, and it is an easy set to understand.

Listing 5.6: Bytecode instructions for test_non_local function

0 LOAD_GLOBAL	0 (0)
3 LOAD_CONST	1 (1)
6 BINARY_POWER	
7 STORE_FAST	0 (0)
10 LOAD_FAST	0 (0)
13 RETURN_VALUE	

Note how rather than a `LOAD_FAST` as was seen in the previous example, we have `LOAD_GLOBAL` instruction. Later, when we discuss the evaluation loop, we will discuss an optimisation that the evaluation loop carries out that makes the use of the `LOAD_FAST` instruction as the name suggests.

10. *co_nlocals*: This is a numeric value that represents the number of local names used by the code object. In the immediate past example from Listing 5.4, the only local variable used is *x* and thus this value is 1 for the code object of that function.
11. *co_stacksize*: The Python virtual machine is stack-based, i.e. values used in evaluation and results of the evaluation are read from and written to an execution stack. This *co_stacksize* value is the maximum number of items that exist on the evaluation stack at any point during the execution of the code block.
12. *co_freevars*: The *co_freevars* field is a collection of free variables defined within the code block. This field is mostly relevant to nested functions that form closures. Free variables are variables that are used within a block but not defined within that block; this does not apply to global variables. The concept of a free variable is best illustrated with an example, as shown in listing 5.7.

Listing 5.7: A simple nested function

```
def f(*args):
    x=1
    def g():
        n = x
```

The *co_freevars* field is empty for the code object of the *f* function while that of the *g* function contains the *x* value. Free variables are strongly interrelated with *cell variables*.

13. *co_cellvars*: The *co_cellvars* field is a collection of names for that require cell storage objects during the execution of a code object. Take the snippet in Listing 5.7, the *co_cellvars* field of the code object for the function - *f*, contains just the name -*x* while that of the nested function's code object is empty; recall from the discussion on free variables that the *co_freevars* collection of the nested function's code object consists of just this name - *x*. This captures the relationship between cell variables, and free variables - a free variable in a nested scope is a cell variable within the enclosing scope. Special cell objects are required to store the values in this cell variable collection during the execution of the code object. This is so because each value in this field is used by nested code objects whose lifetime may exceed that of the enclosing code object. Hence, such values require storage in locations that do not get deallocated after the execution of the code object.

The **bytecode** - *co_code* in more detail.

The actual virtual machine instructions for a code object, the bytecode, are contained in the *co_code* field of a code object as previously mentioned. The byte code from the *fizzbuzz* function, for example, is the string of bytes shown in listing 5.7.

Listing 5.7: Bytecode string for fizzbuzz function

```
b'|\x00d\x01\x16\x00d\x02k\x02r\x1e|\x00d\x03\x16\x00d\x02k\x02r\x1ed\x04S\x00n,|\x0\x0d\x01\x16\x00d\x02k\x02r0d\x05S\x00n\x1a|\x00d\x03\x16\x00d\x02k\x02rBd\x06S\x00n\x\x08t\x00|\x00\x83\x01S\x00d\x00S\x00'
```

To get a human-readable version of the byte string, we use the `dis` function from the `dis` module to extract a human-readable printout as shown in listing 5.8.

Listing 5.8: Bytecode instruction disassembly for fizzbuzz function

7	0 LOAD_FAST	0 (n)
2	2 LOAD_CONST	1 (3)
4	4 BINARY_MODULO	
6	6 LOAD_CONST	2 (0)
8	8 COMPARE_OP	2 (==)
10	10 POP_JUMP_IF_FALSE	30
12	12 LOAD_FAST	0 (n)
14	14 LOAD_CONST	3 (5)
16	16 BINARY_MODULO	
18	18 LOAD_CONST	2 (0)
20	20 COMPARE_OP	2 (==)
22	22 POP_JUMP_IF_FALSE	30
...		
14	>> 66 LOAD_GLOBAL	0 (str)
	68 LOAD_FAST	0 (n)
	70 CALL_FUNCTION	1
	72 RETURN_VALUE	
>>	74 LOAD_CONST	0 (None)
	76 RETURN_VALUE	

The first column of the output shows the line number for that instruction. Multiple instructions may map to the same line number. This value is calculated using information from the `co_lnotab` field of a code object. The second column is the offset of the given instruction from the start of the bytecode. Assuming the bytecode string is contained in an array, then this value is the index of the instruction into the array. The third column is the actual human-readable instruction opcode; the full range of opcodes are found in the `Include/opcode.h` module. The fourth column is the argument to the instruction.

The first `LOAD_FAST` instruction takes the argument `0`. This value is an index into the `co_varnames` array. The last column is the value of the argument - provided by the `dis` function for ease of use. Some arguments do not take explicit arguments. Notice that the `BINARY_MODULO` and `RETURN_VALUE`

instructions take no explicit argument. Recall that the Python virtual machine is stack-based so these instructions read values from the top of the stack.

Bytecode instructions are two bytes in size - one byte for the opcode and the second byte for the argument to the opcode. In the case where the opcode does not take an argument, then the second argument byte is zeroed out. The Python virtual machine uses a little-endian byte encoding on the machine which I am currently typing out this book thus the 16 bits of code are structured as shown in figure 5.0 with the opcode taking up the higher 8 bits and the argument to the opcode taking up the lower 8 bits.



Figure 5.0: Bytecode instruction format showing opcode and oparg

Sometimes, the argument to an opcode may be unable to fit into the default single byte. The Python virtual machine makes use of the EXTENDED_ARG opcode for these kinds of arguments. What the Python virtual machine does is to take an argument that is too large to fit into a single byte and split it into two (we assume that it can fit into two bytes here, but this logic is easily extended past two bytes) - the most significant byte is an argument to the EXTENDED_ARG opcode while the least significant byte is the argument to its actual opcode. The EXTENDED_ARG opcode(s) will come before the actual opcode in the sequence of opcodes, and the argument can then be rebuilt by shifts to the right and **or'ing** with other sections of the argument. For example, if one wanted to pass the value 321 as an argument to the LOAD_CONST opcode, this value cannot fit into a single byte, so the EXTENDED_ARG opcode is used. The binary representation of this value is 0b10100001, so the actual *do work* opcode (LOAD_CONST) takes the first byte (10000001) as argument (65 in decimal) while the EXTENDED_ARG opcode takes the next byte (1) as an argument; thus, we have (144, 1), (100, 65) as the sequence of instructions that is output.

The [documentation for the dis module](#)³ contains a comprehensive list and explanation of all opcodes currently implemented by the virtual machine.

5.2 Code Objects within other code objects

Another code block code object that is worth looking at is that of a module. Assuming we are compiling a module with the `fizzbuzz` function as content, what would the output, look like? To find out, we use the `compile` function in python to compile a module with the content shown in listing 5.9.

³<https://docs.python.org/3.6/library/dis.html>

Listing 5.9: Nested function to illustrated nested code objects

```
def f():
    print(c)
    a = 1
    b = 3
    def g():
        print(a+b)
        c=2
        def h():
            print(a+b+c)
```

Listing 5.10 is the result of compiling a module code block.

Listing 5.10: Bytecode instruction disassembly for listing 5.10

0 LOAD_CONST	0 (<code object f at 0x102a028a0, file "fizzbuzz.py", line 1>)
2 LOAD_CONST	1 ('f')
4 MAKE_FUNCTION	0
6 STORE_NAME	0 (f)
8 LOAD_CONST	2 (None)
10 RETURN_VALUE	

The instruction at byte offset 0 loads a code object stored as the name f - our function definition using the MAKE_FUNCTION Instruction. Listing 5.11 is the content of this code object.

Listing 5.11: Bytecode instruction disassembly for nested function from listing 5.9

```
co_argcount = 0
co_cellvars = ()
co_code = b'd\x00d\x01\x84\x00Z\x00d\x02S\x00'
co_consts = (<code object f at 0x1022029c0, file "fizzbuzz.py", line 1>, 'f', No\
ne)
co_filename = fizzbuzz.py
co_firstlineno = 1
co_flags = 64
co_freevars = ()
co_kwonlyargcount = 0
co_lnotab = b''
co_name = <module>
co_names = ('f',)
co_nlocals = 0
co_stacksize = 2
co_varnames = ()
```

As would be expected in a module, the fields related to code object arguments are all zero - (co_argcount, co_kwonlyargcount). The co_code field contains bytecode instructions, as shown in listing 5.10. The co_consts field is an interesting one. The constants in the field are a code object and the names - f and None. The code object is that of the function, the value 'f' is the name of the function, and None is the return value of the function - recall the python compiler adds a return None statement to a code object without one.

Notice that function objects are not created during the module's compilation. What we have are just code objects - it is during the execution of the code objects that the function gets created as seen in Listing 5.10. Inspecting the attributes of the code object will show that it is also composed of other code objects as shown in listing 5.12.

Listing 5.12: Bytecode instruction disassembly for nested function from listing 5.10

```

co_argcount = 0
co_cellvars = ('a', 'b')
co_code = b't\x00t\x01\x83\x01\x01\x00d\x01\x89\x00d\x02\x89\x01\x87\x00\x87\x01\
f\x02d\x03d\x04\x84\x08}\x00d\x00S\x00'
co_consts = (None, 1, 3, <code object g at 0x101a028a0, file "fizzbuzz.py", line\
5>, 'f.<locals>.g')
co_filename = fizzbuzz.py
co_firstlineno = 1
co_flags = 3
co_freevars = ()
co_kwonlyargcount = 0
co_lnotab = b'\x00\x01\x08\x01\x04\x01\x04\x01'
co_name = f
co_names = ('print', 'c')
co_nlocals = 1
co_stacksize = 3
co_varnames = ('g', )

```

The same logic explained earlier on applies here with the function object created only during the execution of the code object.

5.3 Code Objects in the VM

Like most built-in types, there is the code type that defines the code object type and the PyCodeObject structure for code objects instances. The code type is similar to other type objects that have been discussed in previous sections, so we do not reproduce it here. Listing 5.13 shows the structures used to represent code objects instances.

Listing 5.13: Code object implementation in C

```

typedef struct {
    PyObject_HEAD
    int co_argcount;           /* #arguments, except *args */
    int co_kwonlyargcount;    /* #keyword only arguments */
    int co_nlocals;          /* #local variables */
    int co_stacksize;         /* #entries needed for evaluation stack */
    int co_flags;             /* CO_..., see below */
    int co_firstlineno;       /* first source line number */
    PyObject *co_code;         /* instruction opcodes */
    PyObject *co_consts;        /* list (constants used) */
    PyObject *co_names;         /* list of strings (names used) */
    PyObject *co_varnames;      /* tuple of strings (local variable names) */
    PyObject *co_freevars;      /* tuple of strings (free variable names) */
    PyObject *co_cellvars;      /* tuple of strings (cell variable names) */

    unsigned char *co_cell2arg; /* Maps cell vars which are arguments. */
    PyObject *co_filename;      /* unicode (where it was loaded from) */
    PyObject *co_name;          /* unicode (name, for reference) */
    PyObject *co_lnotab;         /* string (encoding addr<->lineno mapping) See
                                Objects/lnotab_notes.txt for details. */
    void *co_zombieframe;       /* for optimization only (see frameobject.c) */
    PyObject *co_weakreflist;    /* to support weakrefs to code objects */
    /* Scratch space for extra data relating to the code object._icc_nan
       Type is a void* to keep the format private in codeobject.c to force
       people to go through the proper APIs. */
    void *co_extra;
} PyCodeObject;

```

The fields are almost all the same as those found in a Python code objects except for the `co_stacksize`, `co_flags`, `co_cell2arg`, `co_zombieframe`, `co_weakreflist` and `co_extra`. `co_weakreflist` and `co_extra` are not really interesting fields at this point. The rest of the fields here pretty much serve the same purpose as those in the code object. The `co_zombieframe` is a field that exists for optimisation purposes. It holds a reference to a frame object that was previously used as a context to execute the code object. This is then used as the execution frame when such code object is being re-executed to prevent the overhead of allocating memory for another frame object.

6. Frame Objects

Frame objects provide the contextual environment for executing bytecode instructions. Take the set of bytecode instructions in listing 6.0 for example, LOAD_CONST loads values on to a stack, but it has no notion of where or what this stack is. The code object also has no information on the thread or interpreter state that is vital for execution.

Listing 6.0: A set of bytecode instructions

0 LOAD_CONST	0 (<code object f at 0x102a028a0, file "fizzbuzz.py",\nline 1>)
2 LOAD_CONST	1 ('f')
4 MAKE_FUNCTION	0
6 STORE_NAME	0 (f)
8 LOAD_CONST	2 (None)
10 RETURN_VALUE	

Executing code objects requires another data structure that provides such contextual information, and this is where the frame objects come into play. One can think of the frame object as a container in which the code object is executed - it knows about the code object and has references to data and values required during the execution of some code object. As usual, Python does provide us with some facilities to inspect frame objects using the `sys._getframe()` function, as shown in the Listing 6.1 snippet.

Listing 6.1: Accessing frame objects

```
>>> import sys
>>> f = sys._getframe()
>>> f
<frame object at 0x10073ed48>
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'f' is not defined
>>> dir(f)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
 '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'clear',
 'f_back', 'f_builtins', 'f_code', 'f_globals', 'f_lasti', 'f_lineno',
 'f_locals', 'f_trace']
```

Before a code object can be executed, a frame object within which the execution of such a code object takes place is created. Such a frame object contains all the namespaces required for the execution of a code object (*local*, *global*, and *builtin*), a reference to the current thread of execution, stacks for evaluating byte code and other housekeeping information that are important for executing byte code. To get a better understanding of the frame object, let us look at the definition of the frame object data structure from the `Include/frame.h` module and reproduced in listing 6.2.

Listing 6.2: Frame object definition in the vm

```

typedef struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;           /* previous frame, or NULL */
    PyCodeObject *f_code;            /* code segment */
    PyObject *f_builtins;           /* builtin symbol table (PyDictObject) */
    PyObject *f_globals;            /* global symbol table (PyDictObject) */
    PyObject *f_locals;             /* local symbol table (any mapping) */
    PyObject **f_valuestack;        /* points after the last local */
    PyObject **f_stacktop;
    PyObject *f_trace;              /* Trace function */

    /* fields for handling generators*/
    PyObject *f_exc_type, *f_exc_value, *f_exc_traceback;
    /* Borrowed reference to a generator, or NULL */
    PyObject *f_gen;

    int f_lasti;                  /* Last instruction if called */
    int f_lineno;                  /* Current line number */
    int f_iblock;                  /* index in f_blockstack */
    char f_executing;              /* whether the frame is still executing */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
    PyObject *f_localsplus[1];      /* locals+stack, dynamically sized */
} PyFrameObject;

```

The fields coupled with the documentation within the frame are not difficult to understand but we provide a bit more detail about these fields and how they relate to the execution of bytecode.

1. **f_back**: This field is a reference to the frame of the code object that was executing before the current code object. Given a set of frame objects, the **f_back** fields of these frames together form a stack of frames that goes back to the initial frame. This initial frame then has a NULL value in this **f_back** field. This implicit stack of frames forms what we refer to as the **call stack**.
2. **f_code**: This field is a reference to a code object. This code object contains the bytecode that is executed within the context of this frame.

3. `f_builtins`: This is a reference to the `builtin` namespace. This namespace contains names such as `print`, `enumerate` etc. and their corresponding values.
4. `f_globals`: This is a reference to the global namespace of a code object.
5. `f_locals`: This is a reference to the local namespace of a code object. As previously mentioned, these names are defined within the scope of a function. When we discuss the `f_localplus` field, we will see an optimization that Python does when working with locally defined names.
6. `f_valuestack`: This is a reference to the evaluation stack for the frame. Recall that the Python virtual machine is a stack-based virtual machine so, during the evaluation of bytecode, values are read from the top of this stack and results of evaluating the byte code are stored on the top of this stack. This field is the stack that is used during code object execution. The `stacksize` of a frame's code object gives the maximum depth to which this data structure can grow.
7. `f_stacktop`: As the name suggests, the field points to the next free slot of the evaluation value stack. When a frame is newly created, this value is set to the value stack - this is the first available space on the stack as there are no items on the stack.
8. `f_trace`: This field references a function that used for tracing the execution of python code.
9. `f_exc_type`, `f_exc_value`, `f_exc_traceback`, `f_gen`: are fields used for bookkeeping to be able to execute generator code cleanly. More on this when we discuss python generators.
10. `f_localplus`: This is a reference to an array that contains enough space for storing `cell` and `local` variables. This field enables the evaluation loop to optimize loading and storing values of names to and from the value stack with the `LOAD_FAST` and `STORE_FAST` instructions. The `LOAD_FAST` and `STORE_FAST` opcodes provide faster name access than their counterpart `LOAD_NAME` and `STORE_NAME` opcodes because they use array indexing for accessing the value of names and this is done in approximately constant time, unlike their counterparts that search a mapping for a given name. When we discuss the evaluation loop, we see how this value is set up during the frame bootstrapping process.
11. `f_blockstack`: This field references a data structure that acts as a stack used to handle loops and exception handling. This is the second stack in addition to the value stack that is of utmost importance to the virtual machine, but this does not receive as much attention as it rightfully should. The relationship between the block stack, exceptions and looping constructs is quite complicated, and we look at that in the coming chapters.

6.1 Allocating Frame Objects

Frame objects are ubiquitous during python code evaluation - every executed code block needs a frame object that provides some context. New frame objects are created by invoking the `PyFrame_New` function in the `Objects/frameobject.c` module. This function is invoked so many times - whenever a code object is executed, that two main optimizations are used to reduce the overhead of invoking this function, and we briefly look at these optimizations.

First, code objects have a field, the `co_zombieframe` which references an *inert* frame object. When a code object is executed, the frame within which it was executed is not immediately deallocated. The frame is rather maintained in the `co_zombieframe` so when next the same code object executed,

time is not spent allocating memory for a new execution frame. The `ob_type`, `ob_size`, `f_code`, `f_valuestack` fields retain their value; `f_locals`, `f_trace`, `f_exc_type`, `f_exc_value`, `f_exc_traceback` are `NULL` and `f_localplus` retains its allocated space but with the local variables nulled out. The remaining fields do not hold a reference to any object. The second optimization that is used by the virtual machine is to maintain a *free list* of pre-allocated frame objects from which frames can be obtained for the execution of code objects.

The source code for frame objects is a gentle read and one can see how the `zombie` frame and *freelist* concepts are implemented by looking at how allocated frames are deallocated after the execution of the enclosed code object. The interesting part of the code for frame deallocation is shown in listing 6.3.

Listing 6.3: Deallocating frame objects

```

if (co->co_zombieframe == NULL)
    co->co_zombieframe = f;
else if (numfree < PyFrame_MAXFREELIST) {
    ++numfree;
    f->f_back = free_list;
    free_list = f;
}
else
    PyObject_GC_Del(f);

```

Careful observation shows that the *freelist* will only ever grow when a recursive call is made, i.e. a code object tries to execute itself as that is the only time the `zombieframe` field is `NULL`. This small optimization of using the *freelist* helps eliminate to a certain degree, the repeated memory allocations for such recursive calls.

This chapter covers the main points about the frame object without delving into the evaluation loop, which is tightly integrated with the frame objects. A few things left out of this chapter are covered in subsequent chapters. For example,

1. How are values passed on from one frame to the next when code execution hits a `return` statement?
2. What is the thread state, and where does the thread state come from?
3. How are exceptions bubble down the stack of frames when an exception is thrown in the executing frame? Etc.

Most of these question are answered when we look at the interpreter and thread state data structures in the next chapter, and then the evaluation loop in subsequent chapters.

7. Interpreter and Thread States

As mentioned in previous chapters, initializing the interpreter and thread state data structures is one of the steps involved in the bootstrap of the Python interpreter. In this chapter, we provide a detailed look at these data structures.

7.1 The Interpreter state

The `Py_Initialize` function in the `pylifecycle.c` module is one of the bootstrap functions invoked during the initialization of the Python interpreter. This function handles the set-up of the Python runtime as well as the initialization of the interpreter state and thread state data structures among other things.

The interpreter state is a straightforward data structure that captures the global state shared by a set of cooperating threads of execution in a Python process. Listing 7.0 is a cross-section of this data structure's definition.

Listing 7.0: Cross-section of the interpreter state data structure

```
typedef struct _is {

    struct _is *next;
    struct _ts *tstate_head;

    PyObject *modules;
    PyObject *modules_by_index;
    PyObject *sysdict;
    PyObject *builtins;
    PyObject *importlib;

    PyObject *codec_search_path;
    PyObject *codec_search_cache;
    PyObject *codec_error_registry;
    int codecs_initialized;
    int fscodec_initialized;

    ...
    PyObject *builtins_copy;
```

```
PyObject *import_func;
} PyInterpreterState
```

The fields in listing 7.0 should be familiar if one covered the prior materials in this book, and has used Python for a considerable amount of time. We discuss some of the fields of the interpreter state data structure once again.

- `*next`: There can be multiple interpreter states within a single OS process that is running a python executable. This `*next` field references another interpreter state data structure within the python process if such exist, and these form a linked list of interpreter states, as shown in figure 7.0. Each interpreter state has its own set of variables that will be used by a thread of execution that references that interpreter state. However, all interpreter threads in the process share the same memory space and *Global Interpreter Lock*.

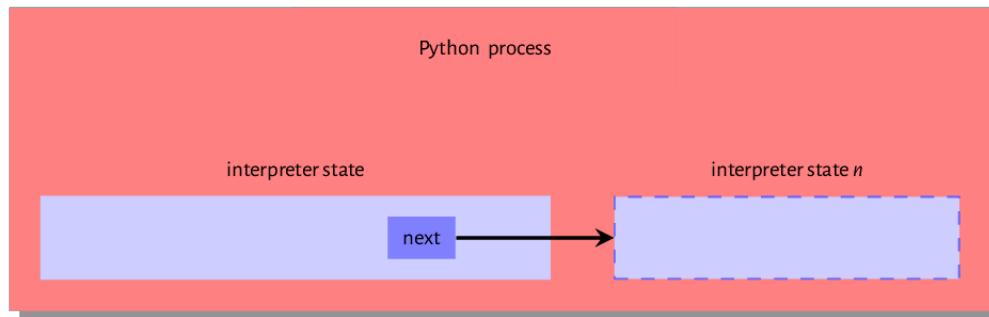


Figure 7.0: Interpreter state within the executing python process

- `*tstate_head`: This field references the thread state of the currently executing thread or in the case of a multithreaded program, the thread that currently holds the Global Interpreter Lock (GIL). This is a data structure that maps to an executing operating system thread.

The remaining fields are variables that are shared by all cooperating threads of the interpreter state. The `modules` field is a table of installed Python modules - we see how the interpreter finds these modules later on when we discuss the import system, the `builtins` field is a reference to the built-in `sys` module. The content of this module is the set of built-in functions such as `len`, `enumerate` etc. and the `Python/builtinmodule.c` module contains implementations for most of the contents of the module. The `importlib` is a field that references the implementation of the import mechanism - we speak a bit more about this when we discuss the import system in detail. The `*codec_search_path`, `*codec_search_cache`, `*codec_error_registry`, `*codecs_initialized` and `*fscodec_initialized` are fields that all relate to codecs that Python uses to encode and decode bytes and text. The interpreter uses values in these fields to locate such codecs as well as handle errors related to using such codecs. An executing Python program is composed of one or more threads of execution. The interpreter has to maintain some state for each thread of execution, and this works by maintaining a thread state data structure for each thread of execution. We look at this data structure next.

7.2 The Thread state

Reviewing the *Thread state* data structure in listing 7.1, one can see that the thread state data structure is a more involved data structure than the interpreter state data structure.

Listing 7.2: Cross-section of the thread state data structure

```

typedef struct _ts {
    struct _ts *prev;
    struct _ts *next;
    PyInterpreterState *interp;

    struct _frame *frame;
    int recursion_depth;
    char overflowed;
    char recursion_critical;
    int tracing;
    int use_tracing;

    Py_tracefunc c_profilefunc;
    Py_tracefunc c_tracefunc;
    PyObject *c_profileobj;
    PyObject *c_traceobj;

    PyObject *curexc_type;
    PyObject *curexc_value;
    PyObject *curexc_traceback;

    PyObject *exc_type;
    PyObject *exc_value;
    PyObject *exc_traceback;

    ...
}

} PyThreadState;

```

A thread state data structure's previous and next fields reference thread states created before and just after the given thread state. These fields form a doubly-linked list of thread states that share a single interpreter state. The interp field references the thread state's interpreter state. The frame references the current frame of execution; the value referenced by this field changes when the code object that is executing changes.

The recursion_depth, as the name suggests, specifies how deep the stack frame should get during a recursive call. The overflowed flag is set when the stack overflows. After a stack overflow, the

thread allows 50 more calls for clean-up operations. The `recursion_critical` flag signals to the thread that the code being executed should not overflow. The `tracing` and `use_tracing` flag are related to functionality for tracing the execution of the thread. The `*curexc_type`, `*currexc_value`, `*curexc_traceback`, `*exc_type`, `*exc_value` and `*curexc_traceback` are fields that are all used in the exception handling process as will be seen in subsequent chapters.

It is essential to understand the difference between the thread state and an actual thread. The thread state is just a data structure that encapsulates some state for an executing thread. Each thread state is associated with a native OS thread within the Python process. Figure 7.1 is an excellent visual illustration of this relationship. We can see that a single python process is home to at least one interpreter state and each interpreter state is home to one or more thread states, and each of these thread states maps to an operating system thread of execution.

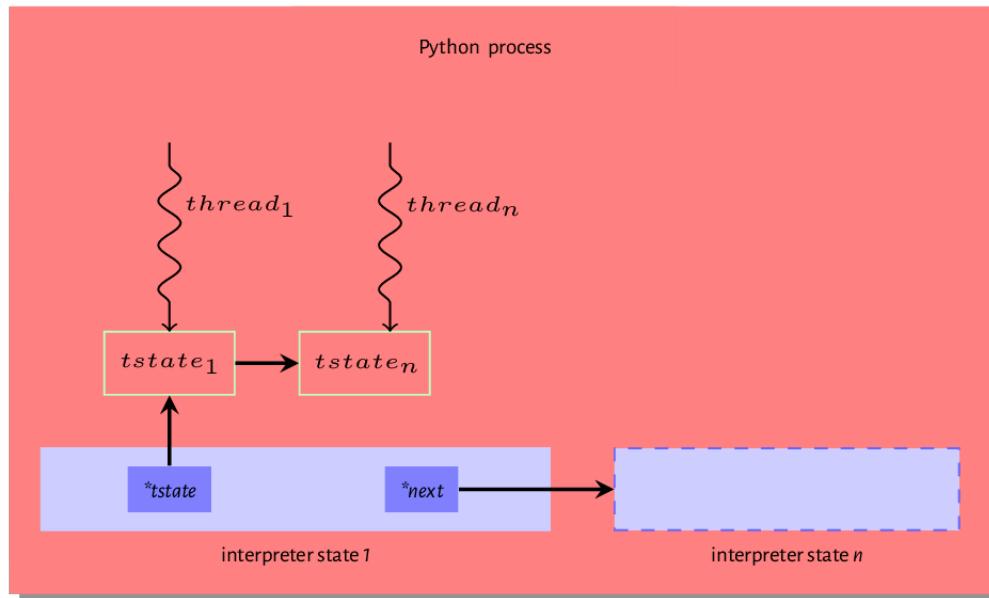


Figure 7.1: Relationship between interpreter state and thread states

Operating System threads and associated Python thread states are created either during the initialization of the interpreters or when invoked by the threading module. Even with multiple threads alive within a Python process, only one thread can actively carry out CPU bound tasks at any given time. This is because an executing thread must hold the GIL to execute byte code within the python virtual machine. This chapter will not be complete without a look at the famous or *infamous* GIL concept so we take this on in the next section

Global Interpreter Lock - GIL

Although python threads are operating system threads, a thread cannot execute python bytecode unless such thread holds the GIL. The operating system may schedule a thread that does not hold the GIL to run but as we will see, all such a thread can do is wait to get the GIL and only when it holds the GIL is it able to execute bytecode. We take a look at this whole process.

The Need for a GIL

Before we begin any discussion on the GIL, it is worthwhile to ask why we need a global lock that will probably adversely affects our threads? The GIL is relevant for a myriad of reasons. First of all, however, it is important to understand that the GIL is an implementation detail of CPython and not an actual language detail - Jython which is Python implemented on the Java virtual machine has no notion of a GIL. The primary reason the GIL exist is for ease of implementation of the CPython virtual machine. It is way easier to implement a single global lock than to implement fine-grained locks and the core developers have opted for this. There have however been projects to implement fine-grained locks within the Python virtual machine but these have sometimes slowed down single-threaded programs. A global lock also provides much-needed synchronization when performing specific tasks. Take the reference counting mechanism that is used by CPython for memory management, without the concept of a GIL, you may have two thread interleave their increment and decrement of reference count, leading to severe issues with memory handling. Another reason for this lock is that some C libraries that CPython calls into are inherently not thread-safe, so some kind of synchronization is required when using them.

When the interpreter startups, a single main thread of execution is created, and there is no contention for the GIL as there is no other thread around, so the main thread does not bother to acquire the lock. The GIL comes into play after other threads are spawned. The snippet in listing 7.3 is from the `Modules/_threadmodule.c` and provides insight into this process during the creation of a new thread.

Listing 7.3: Cross-section of code for creating new thread

```

boot->interp = PyThreadState_GET()->interp;
boot->func = func;
boot->args = args;
boot->keyw = keyw;
boot->tstate = _PyThreadState_Prealloc(boot->interp);
if (boot->tstate == NULL) {
    PyMem_DEL(boot);
    return PyErr_NoMemory();
}
Py_INCREF(func);
Py_INCREF(args);
Py_XINCREF(keyw);
PyEval_InitThreads(); /* Start the interpreter's thread-awareness */
ident = PyThread_start_new_thread(t_bootstrap, (void*) boot);

```

The snippet in listing 7.3 is from the `thread_PyThread_start_new_thread` function that is invoked to create a new thread. `boot` is a data structure the contains all the information that a new thread needs

to execute. The `tstate` field references the thread state for the new thread, and the `_PyThreadState_Prealloc` function call creates this thread state. The main thread of execution must acquire the GIL before creating the new thread; a call to `PyEval_InitThreads` handles this. With the interpreter now thread-aware and the main thread holding the GIL, the `PyThread_start_new_thread` is invoked to create the new operating system thread. The `_tbootstrap` function in the `Modules/_threadmodule.c` module is a callback function invoked by new threads when they come alive. A snapshot of this bootstrap function is in listing 7.4.

Listing 7.4: Cross-section of thread bootstrapping function

```

static void t_bootstrap(void *boot_raw){
    struct bootstate *boot = (struct bootstate *) boot_raw;
    PyThreadState *tstate;
    PyObject *res;

    tstate = boot->tstate;
    tstate->thread_id = PyThread_get_thread_ident();
    _PyThreadState_Init(tstate);
    PyEval_AcquireThread(tstate);
    nb_threads++;
    res = PyEval_CallObjectWithKeywords(
        boot->func, boot->args, boot->keyw);

    ...
}

```

Notice the call to `PyEval_AcquireThread` function in listing 7.4. The `PyEval_AcquireThread` function is defined in the `Python/ceval.c` module and it invokes the `take_gil` function, which is the function that attempts to get a hold of the GIL. A description of this process, as provided in the source file, is quoted in the following text.

The GIL is just a boolean variable (`gil_locked`) whose access is protected by a mutex (`gil_mutex`), and whose changes are signalled by a condition variable (`gil_cond`). `gil_mutex` is taken for short periods of time, and therefore mostly uncontended. In the GIL-holding thread, the main loop (`PyEval_EvalFrameEx`) must be able to release the GIL on demand by another thread. A volatile boolean variable (`gil_drop_request`) is used for that purpose, which is checked at every turn of the eval loop. That variable is set after a wait of `interval` microseconds on `gil_cond` has timed out. [Actually, another volatile boolean variable (`eval_breaker`) is used which ORs several conditions into one. Volatile booleans are sufficient as inter-thread signalling means since Python is run on cache-coherent architectures only.] A thread wanting to take the GIL will first let pass a given amount of time (`interval` microseconds) before setting `gil_drop_request`. This encourages a defined switching period, but does not enforce it since opcodes can take an arbitrary time to execute. The `interval` value is available for the user to read and modify using the

Python API `sys.{get, set}switchinterval()`. When a thread releases the GIL and `gil_drop_request` is set, that thread ensures that another GIL-awaiting thread gets scheduled. It does so by waiting on a condition variable (`switch_cond`) until the value of `gil_last_holder` is changed to something else than its own thread state pointer, indicating that another thread has taken GIL. This prohibits the latency-adverse behaviour on multi-core machines where one thread would speculatively release the GIL, but still, run and end up being the first to re-acquire it, making the “timeslices” much longer than expected.

What does the above mean for a newly spawned thread? The `t_bootstrap` function in listing 7.4 invokes the `PyEval_AcquireThread` function that handles *requesting* for the GIL. A lay explanation for what happens during this request is thus - assume A is the main thread of execution holding the GIL while B is the new thread that has just been spawned.

1. When B is spawned, `take_gil` is invoked. This checks if the conditional `gil_cond` variable is set. If it is not set then the thread starts a wait.
2. After wait time elapses, the `gil_drop_request` is set.
3. Thread A, after each trip through the execution loop, checks if any other thread has set the `gil_drop_request` variable.
4. Thread A drops the GIL when it detects that the `gil_drop_request` variable is set and also sets the `gil_cond` variable.
5. Thread A also waits on another variable - `switch_cond`, until the value of the `gil_last_holder` is set to a value other than thread A’s thread state pointer indicating that another thread has taken the GIL.
6. Thread B now has the GIL and can go ahead to execute bytecode.
7. Thread A waits a given time, sets the `gil_drop_request` and the cycle continues.

GIL and Performance

The GIL is the primary reason why increasing the number of threads working on a CPU bound program in a single processor core setting in in Python most times does not speed up such a program. In fact, at times, adding a thread will adversely affect the performance of a program when compared to that of a single-threaded program; this is because there is a cost associated with all the switches and waits.

To conclude this chapter, we recap the model of the Python virtual machine we have created so far that captures when we run the Python interpreter with a source file. First, the interpreter initializes interpreter and thread states, the source in the file is compiled into a code object. The code object is then passed to the interpreter loop where a frame object is created and attached to the main thread of execution, so the execution of the code object can happen. So we have a Python process that may contain one or more interpreter states, and each interpreter state may have one or more thread

states, and each thread states references a frame that may reference another frame etc., forming a stack of frames. Figure 7.2 provides a visual representation of this order.

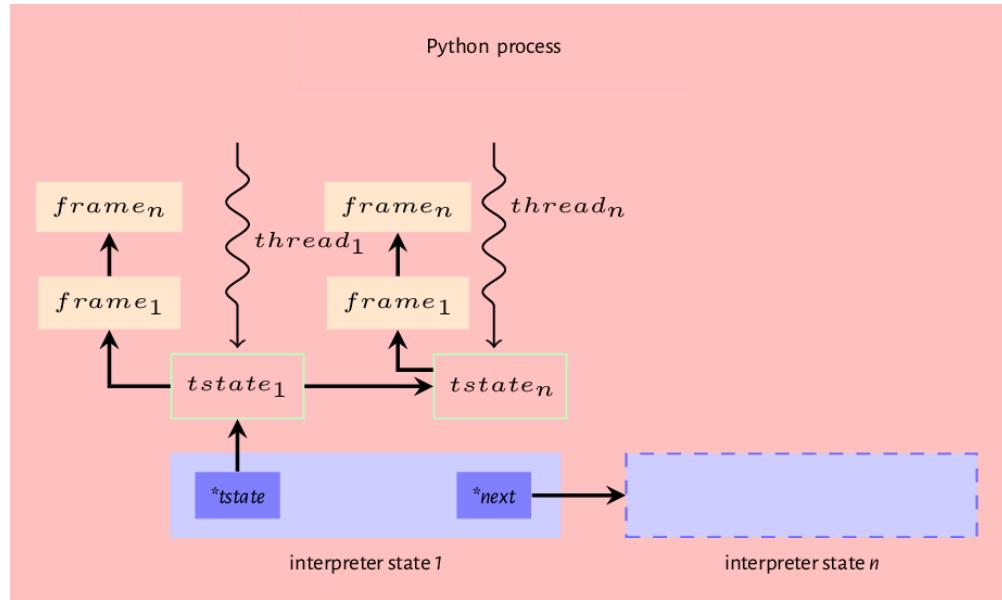


Figure 7.2: Interpreter state, thread state and frame relationship

In the next chapter, we show how all the parts that we have described enable the execution of a Python code object.

8. Intermezzo: The `abstract.c` Module

We have thus far mentioned severally that the Python virtual machine generically treat values for evaluation as `PyObjects`. This begets the obvious question - *How are operations safely carried out on such generic objects?* For example, when evaluating the bytecode instruction `BINARY_ADD`, two `PyObject` values are popped from the evaluation stack and used as arguments to an add operation, but how does the virtual machine know if the add operation makes sense for both values?

To understand how a lot of the operations on `PyObject`s work, we only have to look at the `Objects/Abstract.c` module. This module defines several functions that operate on objects that implement a given object protocol. This means that for example, if one were adding two objects, then the add function in this module would expect that both objects implement the `__add__` method of the `tp_numbers` slots. The best way to explain this is to illustrate with an example.

Consider when the `BINARY_ADD` opcode adds two numbers, the function that does the addition is the `PyNumber_Add` function of the `Objects/Abstract.c` module. Listing 8.1 is the definition of the `PyNumber_Add` function.

Listing 8.1: Generic add function from `abstract.c` module

```
1  PyObject * PyNumber_Add(PyObject *v, PyObject *w){
2      PyObject *result = binary_op1(v, w, NB_SLOT(nb_add));
3      if (result == Py_NotImplemented) {
4          PySequenceMethods *m = v->ob_type->tp_as_sequence;
5          Py_DECREF(result);
6          if (m && m->sq_concat) {
7              return (*m->sq_concat)(v, w);
8          }
9          result = binop_type_error(v, w, "+");
10     }
11     return result;
12 }
```

Our interest at this point is in line 2 of the `PyNumber_Add` function defined in listing 8.1 - the call to the `binary_op1` function. The `binary_op1` function is another generic function that takes among its parameters, two numbers or subclass of numbers and applies a binary function to these; the `NB_SLOT` macro returns the offset of a given method into the `PyNumberMethods` structure; recall that this structure is a collection of methods that work on numbers. The definition of this generic `binary_op1` function is in listing 8.2, and an in-depth explanation of this function immediately follows.

Listing 8.2: The generic binary_op1 function

```

1  static PyObject * binary_op1(PyObject *v, PyObject *w, const int op_slot){
2      PyObject *x;
3      binaryfunc slotv = NULL;
4      binaryfunc slotw = NULL;
5
6      if (v->ob_type->tp_as_number != NULL)
7          slotv = NB_BINOP(v->ob_type->tp_as_number, op_slot);
8      if (w->ob_type != v->ob_type &&
9          w->ob_type->tp_as_number != NULL) {
10         slotw = NB_BINOP(w->ob_type->tp_as_number, op_slot);
11         if (slotw == slotv)
12             slotw = NULL;
13     }
14     if (slotv) {
15         if (slotw && PyType_IsSubtype(w->ob_type, v->ob_type)) {
16             x = slotw(v, w);
17             if (x != Py_NotImplemented)
18                 return x;
19             Py_DECREF(x); /* can't do it */
20             slotw = NULL;
21         }
22         x = slotv(v, w);
23         if (x != Py_NotImplemented)
24             return x;
25         Py_DECREF(x); /* can't do it */
26     }
27     if (slotw) {
28         x = slotw(v, w);
29         if (x != Py_NotImplemented)
30             return x;
31         Py_DECREF(x); /* can't do it */
32     }
33     Py_RETURN_NOTIMPLEMENTED;
34 }
```

1. The function takes three values, two `PyObject *` - `v` and `w` and an integer value, operation slot, which is the offset of that operation into the `PyNumberMethods` structure.
2. Lines 3 and 4 define two values `slotv` and `slotw`, structures that represent a binary function as their types suggest.

3. From line 3 to line 13, we attempt to dereference the function given by `op_slot` argument for both `v` and `w`. On line 8, there is an equality check of both values' types, and if they are of the same type, there is no need to dereference the second value's function in the `op_slot`. If both values are not of the same type, but the functions dereferenced from both are equal, then the `slotw` value is nulled out.
4. With the binary functions dereferenced, if `slotv` is not `NULL` then on line 15 we check that `slotw` is not `NULL` and the type of `w` is a subtype of the type of `v`. If that is true, `slotw`'s function is applied to both `v` and `w`. This happens because if you pause to think about it for a second, the method further down the inheritance tree is what we want to use not one further up. If `w` is not a subtype, then `slotv` is applied to both values at line 22.
5. Getting to line 27 means that the `slotv` function is `NULL` so we apply whatever `slotw` references to both `v` and `w` so long as it is not `NULL`.
6. In the case where both `slotv` and `slotw` both do not contain a function, then a `Py_NotImplemented` is returned. `Py_RETURN_NOTIMPLEMENTED` is just a macro that increments the reference count of the `Py_NotImplemented` value before returning it.

The idea captured by the explanation given above is a blueprint for how the interpreter performs operations on values. We have simplified things a bit here by ignoring that opcodes that can be overloaded. For example, the `+` symbol maps to the `BINARY_ADD` opcode and applies to strings, numbers and some sequences, but we have only looked at it in the context of numbers and subclasses of numbers. The `BINARY_ADD` implementation shown in Listing 8.3 can handle the other cases by looking at the type of values it is operating on and calling the corresponding functions. First, if both values are Unicode characters, the interpreter calls the function for concatenating Unicode characters. Otherwise, the `PyNumber_Add` function is invoked. This function's implementation shows how it checks for numeric and then sequence types applying corresponding addition functions to the different types.

Listing 8.3: ceval implementation of binary add

```

PyObject *right = POP();
PyObject *left = TOP();
PyObject *sum;
if (PyUnicode_CheckExact(left) &&
    PyUnicode_CheckExact(right)) {
    sum = unicode_concatenate(left, right, f, next_instr);
    /* unicode_concatenate consumed the ref to left */
}
else {
    sum = PyNumber_Add(left, right);
    Py_DECREF(left);
}

```

Ignore lines 1 and 2 as we discuss them when we talk about the interpreter loop. What we see from the rest of the snippet, is that when we encounter the `BINARY_ADD`, the first port of call is a check that

both values are strings to apply string concatenation to the values. The `PyNumber_Add` function from `Objects/Abstract.c` is then applied to both values if they are not strings. Although the code seems a bit messy with the string check done in `Python/ceval.c` and the number and sequence checks done in `Objects/Abstract.c`, it is pretty clear what is happening when we have an overloaded opcode.

This explanation provided above is the way the interpreter handles most opcode operations - check the types of the values then dereference the method as required and apply to the argument values.

9. The evaluation loop, `ceval.c`

We have finally arrived at the gut of the virtual machine where the virtual machine iterates over a code object's bytecode instructions and executes such instructions. The essence of this is a `for` loop that iterates over opcodes, switching on each opcode type to run the desired code. The `Python/ceval.c` module, about 5411 lines long, implements most of the functionality required - at the heart of this function is the `PyEval_EvalFrameEx` function, an approximately 3000 line long function that contains the actual evaluation loop. It is this `PyEval_EvalFrameEx` function that is the main thrust of our focus in the chapter.

The `Python/ceval.c` module provides platform-specific optimizations such as *threaded gotos* as well as Python virtual machine optimizations such as opcode prediction. In this write-up, we are more concerned with the virtual machine processes and optimizations, so we conveniently disregard any platform-specific optimizations or process introduced here so long as it does not take away from our explanation of the evaluation loop. We go into more detail than usual here to provide a solid explanation for how the heart of the virtual machine is structured and works. It is important to note that the opcodes and their implementations are constantly in flux so this description here may be inaccurate at a later time.

Before any bytecode execution happens, several housekeeping operations such as creating and initializing frames, setting up variables and initializing the virtual machine variables such as instruction pointers are carried out. We look at some of these operations next.

9.1 Putting names in place

As mentioned above, the heart of the virtual machine is the `PyEval_EvalFrameEx` function that executes the Python bytecode but before this happens, a lot of setups - error checking, frame creation and initialization etc. - need to take place to prepare the evaluation context. This is where the `_PyEval_EvalCodeWithName` function also within the `Python/ceval.c` module comes in. For illustration purposes, we assume that we are working with a module that has the content shown in listing 9.0.

Listing 9.0: Content of a simple module

```

def test(arg, defarg="test", *args, defkwd=2, **kwd):
    local_arg = 2
    print(arg)
    print(defarg)
    print(args)
    print(defkwd)
    print(kwd)

test()

```

Recall that code blocks have code objects; these code blocks could either be functions, modules etc. so for a module with the above content, we can safely assume that we are dealing with two code objects - one for the module and one for the function `test` defined within the module.

After the generation of the code object for the module in listing 9.0, the generated code object is executed via a chain of function calls from the `Python/pythonrun.c` module - `run_mod -> PyEval_EvalCode -> PyEval_EvalCodeEx -> _PyEval_EvalCodeWithName -> PyEval_EvalFrameEx`. At this moment, our interest lies with the `_PyEval_EvalCodeWithName` function with its signature shown in listing 9.1. It handles the required name setup before bytecode evaluation in `PyEval_EvalFrameEx`. However, by looking at the function signature for the `_PyEval_EvalCodeWithName` as shown in listing 9.1, one is probably left asking how this is related to executing a module object rather than an actual function.

Listing 9.1: `_PyEval_EvalCodeWithName` function signature

```

static PyObject * _PyEval_EvalCodeWithName(PyObject * _co, PyObject * globals, PyObject *locals,
                                         PyObject **args, int argcount, PyObject **kws, int kwcount,
                                         PyObject **defs, int defcount, PyObject *kwdefs, PyObject *closure,
                                         PyObject *name, PyObject *qualname)

```

To wrap one's head around this, one must think more generally in terms of code blocks and code objects, not functions or modules. Code blocks can have any or none of those arguments specified in the `_PyEval_EvalCodeWithName` function signature - a function just happens to be a more specific type of code block which has most if not all those values supplied. This means that the case of executing `_PyEval_EvalCodeWithName` for a module code object is not very interesting as most of those arguments are without value. The interesting instance occurs when a Python function call is made via the `CALL_FUNCTION` opcode. This results in a call to the `fast_function` function also in the `Python/ceval.c` module. This function extracts function arguments from the function object before delegating to the `_PyEval_EvalCodeWithName` function to carry out all the sanity checks that are needed - this is not the full story, but we will look at the `CALL_FUNCTION` opcode in more detail in a later section of this chapter.

The `_PyEval_EvalCodeWithName` is quite a big function, so we do not include it here, but most of the setup process that it goes through is pretty straightforward. For example, recall we mentioned that the `fastlocals` field of a frame object provides some optimization for the local namespace and that non-positional function arguments are known fully only at runtime. This means that we cannot populate this `fastlocals` data structure without careful error checking. It is during this setup by the `_PyEval_EvalCodeWithName` function that the array referenced by the `fastlocals` field of a frame is populated with the full range of local values. The steps involved in the setup process that the `_PyEval_EvalCodeWithName` goes through when called involves the steps shown in listing 9.1.

Listing 9.2: `_PyEval_EvalCodeWithName` setup steps

-
1. Initialize a frame object that provides context for the code object execution.
 3. Add the keyword `*dict*` to the frame `fast locals`.
 4. Add positional arguments to `'fastlocals'`.
 5. Add the variable sequence of non-positional, non-keyword arguments to the `'fastlocals'` array (`'*args'` from our example module). These values are held together in a tuple data structure.
 6. Check that any keyword argument supplied to a code block is expected and has not \ been provided twice.
 7. Check for missing positional arguments and throw an error if any are found.
 8. Add the default arguments to the `'fastlocals'` array (`'defarg'` in our example modu\le).
 9. Add keyword defaults to `'fastlocals'` (`'defkwd'` in our example module).
 10. Initialize storage for cell variables and copy free variables array into the frame.
 11. Do some generator related housekeeping - we look at this in more detail when we discuss generators.
-

9.2 The parts of the machine

With all the names in place, `PyEval_EvalFrameEx` is invoked with a frame object as one of its arguments. A cursory look at this function shows that the function is composed of quite a few C macros and variables. The *macros* are an integral part of the execution loop - they provide a means to abstract away repetitive code without incurring the cost of a function call and as such we describe a few of them. In this section, we assume that the virtual machine is not running with C optimizations such as `computed gotos` enabled so we conveniently ignore macros related to such optimizations.

We begin with a description of some of the variables that are crucial to the execution of the evaluation loop.

1. `**stack_pointer`: refers to the next free slot in the value stack of the execution frame.

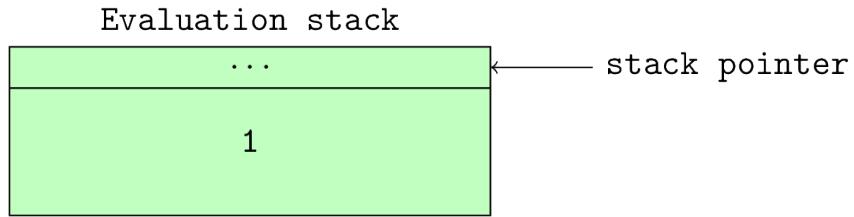


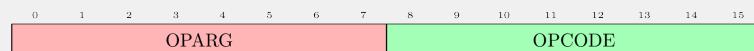
Figure 9.0: Stack pointer after a single value is pushed onto the stack

1. `*next_instr`: refers to the next instruction to be executed by the evaluation loop. One can think of this as the *program counter* for the virtual machine. Python 3.6 changes the type of this value to an `unsigned short` which is 2 bytes in size to handle the new bytecode instruction size.
2. `opcode`: refers to the currently executing python opcode or the opcode that is about to be executed.
3. `oparg`: refers to the argument of the presently executing opcode or opcode that is about to be executed if it takes an argument.
4. `why`: The evaluation loop is an infinite loop implemented by the infinite `for` loop - `for(;;)` so the loop needs a mechanism to break out of the loop and specify why the break occurred. This value refers to the reason for an exit from the evaluation loop. For example if the code block exited the loop due to a `return` statement, then this value will contain a `WHY_RETURN` status.
5. `fastlocals`: refers to an array of locally defined names.
6. `freevars`: refers to a list of names that are used within a code block but not defined in that code block.
7. `retval`: refers to the return value from executing the code block.
8. `co`: References the code object that contains the bytecode that will be executed by the evaluation loop.
9. `names`: This references the names of all values in the code block of the executing frame.
10. `consts`: This references the constants used by the code objects.

Bytecode instruction

We have discussed the format of bytecode instructions in the chapter on code objects, but it is very relevant to our discussion here so we repeat our description of the bytecode instruction format here.

Assuming we are working with python 3.6 bytecodes, all bytecodes are 16 bit long. The Python VM uses a little-endian byte encoding on the machine which I am currently typing out this book thus the 16 bits of code are structured as shown in the following image with the opcode taking up 1 byte and the argument to the opcode taking up the second byte.



Bytecode instruction format showing oparg and opcode

Extracting the opcodes and arguments involves some bit manipulation as we will see in the following sections. It is important to note that since the opcode is now two bytes and not one, then the manipulation of instruction pointers subscribes to [pointer manipulation^a](#).

^a<https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/pointer.html>

The following macros play a vital role in the evaluation loop.

1. `TARGET(op)`: expands to the case `op` statement. This matches the current opcode with the block of code that implements the opcode.
2. `DISPATCH`: expands to `continue`. This together with the next macro - `FAST_DISPATCH`, handle the flow of control of the evaluation loop after an opcode is executed.
3. `FAST_DISPATCH`: expands to a jump to the `fast_next_opcode` label within the evaluation for loop.

With the introduction of the standard 2 bytes opcode in Python 3.6, the following set of macros are used to handle code access.

1. `INSTR_OFFSET()`: This macro provides the byte offset of the current instruction into the array of instructions. This expands to `(2*(int)(next_instr - first_instr))`.
2. `NEXTOPARG()`: This updates the `opcode` and `oparg` variable to the value of the opcode and argument of the next bytecode instruction to be executed. This macro expands to the following snippet.

Listing 9.3: Expansion of the `NEXTOPARG` macro

```
do { \
    unsigned short word = *next_instr; \
    opcode = OPCODE(word); \
    oparg = OPARG(word); \
    next_instr++; \
} while (0)
```

The `OPCODE` and `OPARG` macros handle the bit manipulation for extracting opcode and arguments. Figure 9.0 shows the structure of a bytecode instruction with the argument to the opcode taking lower eight bits and the opcode itself taking the upper eight bits hence `OPCODE` expands to `((word) & 255)` thus extracting the most significant byte from the bytecode instruction while `OPARG` which expands to `((word) >> 8)` extracts the least significant byte.

1. `JUMPTO(x)`: This macro expands to `(next_instr = first_instr + (x)/2)` and performs an absolute jump to a particular offset in the bytecode stream.

2. `JUMPBY(x)`: This macro expands to `(next_instr += (x)/2)` and performs a relative jump from the current instruction offset to another point in the bytecode instruction stream.
3. `PREDICT(op)`: This opcode together with the `PREDICTED(op)` opcode implement the Python evaluation loop opcode prediction. This opcode expands to the following snippet.

Listing 9.4: Expansion of the `PREDICT(op)` macro

```

do{ \
    unsigned short word = *next_instr; \
    opcode = OPCODE(word); \
    if (opcode == op){ \
        oparg = OPARG(word); \
        next_instr++; \
        goto PRED_##op; \
    } \
} while(0)

```

4. `PREDICTED(op)`: This macro expands to `PRED_##op:.`

The last two macros defined above handle opcode prediction. When the evaluation loop encounters a `PREDICT(op)` macro; the interpreter assumes that the next instruction to be executed is `op`. The macros check that this is indeed valid and if valid fetches the actual opcode and argument then jumps to the label `PRED_##op` where the `##` is a placeholder for the actual opcode. For example, if we had encountered a prediction such as `PREDICT(LOAD_CONST)` then the `goto` statement argument would be `PRED_LOAD_CONST` if that prediction was valid. An inspection of the source code for the `PyEval_EvalFrameEx` function finds the `PREDICTED(LOAD_CONST)` label that expands to `PRED_LOAD_CONST` so on a successful prediction of this instruction; there is a jump to this label otherwise normal execution continues. This prediction saves the cost involved with the extra traversal of the `switch` statement that would otherwise happen with normal code execution.

The next set of macros that we are interested in are the stack manipulation macros that handle placing and fetching of values from the value stack of a frame object. These macros are pretty similar, and the following snippet shows a few examples.

1. `STACK_LEVEL()`: This returns the number of items on the stack. The macro expands to `((int)(stack_pointer - f->f_valuestack)).`
2. `TOP()`: The returns the last item on the stack. This expands to `(stack_pointer[-1]).`
3. `SECOND()`: This returns the penultimate item on the stack. This expands to `(stack_pointer[-2]).`
4. `BASIC_PUSH(v)`: This places the item, `v`, on the stack. It expands to `(*stack_pointer++ = (v)).` A current alias for this macro is the `PUSH(v)`.
5. `BASIC_POP()`: This removes and returns an item from the stack. This expands to `(*--stack_pointer).` A current alias for this is the `POP()` macro.

The last set of macros of concern to us are those that handle local variable manipulation. These macros, `GETLOCAL` and `SETLOCAL` are used to get and set values in the `fastlocals` array.

1. `GETLOCAL(i)`: This expands to `(fastlocals[i])`. This handles the fetching of locally defined names from the local array.
2. `SETLOCAL(i, value)`: This expands to the snippet in listing 9.5. This macro sets the *i*th element of the local array to the supplied value.

Listing 9.5: Expansion of the `SETLOCAL(i, value)` macro

```
do { PyObject *tmp = GETLOCAL(i); \
     GETLOCAL(i) = value; \
     Py_XDECREF(tmp); \
} while (0)
```

The `UNWIND_BLOCK` and `UNWIND_EXCEPT_HANDLER` are related to exception handling, and we look at them in subsequent sections.

9.3 The Evaluation loop

We have finally come to the heart of the virtual machine - the loop where the opcodes are evaluated. The implementation is pretty anti-climatic as there is nothing special here, just a *never-ending* `for` loop and a massive `switch` statement that matches on opcodes. To get a concrete understanding of this statement, we look at the execution of the simple hello world function in listing 9.6.

Listing 9.6: Simple hello world python function

```
def hello_world():
    print("hello world")
```

Listing 9.7 shows the disassembly of the function from Listing 9.6, and we illustrate the evaluation of this set of instructions.

Listing 9.7: Disassembly of function in listing 9.6

LOAD_GLOBAL	0 (0)
LOAD_CONST	1 (1)
CALL_FUNCTION	1 (1 positional, 0 keyword pair)
POP_TOP	
LOAD_CONST	0 (0)
RETURN_VALUE	

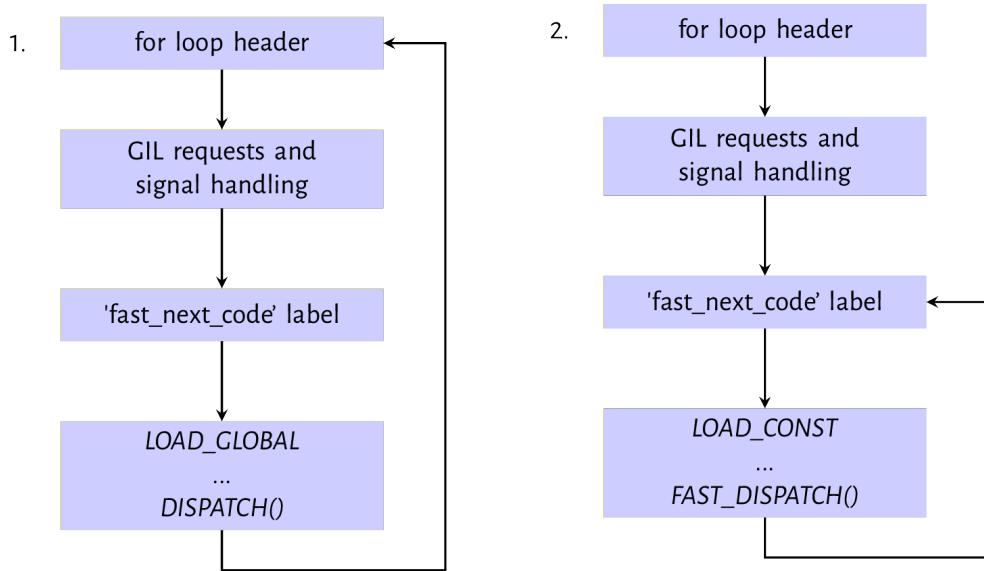
Figure 9.1: Evaluation path for `LOAD_GLOBAL` and `LOAD_CONST` instructions

Figure 9.1 shows the evaluation path for the `LOAD_GLOBAL` and `LOAD_CONST` instructions. The second and third blocks in both images of figure 9.2 represent the housekeeping tasks performed on every iteration of the evaluation loop. The `GIL` and signal handling checks were discussed in the the previous chapter on interpreter and thread states - it is during these checks that a thread executing may give up control of the `GIL` for another thread to execute. The `fast_next_opcode` is a code label just after the `GIL` and signal handling code that exists to serve as a jump destination when the loop wishes to skip the previous checks as we will see when we look at the `LOAD_CONST` instruction.

The first instruction - `LOAD_GLOBAL` is evaluated by the `LOAD_GLOBAL` case statement of the `switch` statement. The implementation of this opcode like other opcodes is a series of C statements and function calls that are surprisingly involved, as shown in listing 9.8. The implementation of the opcode loads the value identified by the given name from the global or builtin namespace onto the evaluation stack. The `oparg` is the index into the tuple which contains all names used within the code block - `co_names`.

Listing 9.8: `LOAD_GLOBAL` implementation

```

PyObject *name = GETITEM(names, oparg);
PyObject *v;
if (PyDict_CheckExact(f->f_globals)
    && PyDict_CheckExact(f->f_builtins)){
    v = _PyDict_LoadGlobal((PyDictObject *)f->f_globals,
                          (PyDictObject *)f->f_builtins,
                          name);
if (v == NULL) {
    if (!PyErr_OCCURRED()) {
        /* _PyDict_LoadGlobal() returns NULL without raising
         * an exception if the key doesn't exist */

```

```

        format_exc_check_arg(PyExc_NameError,
                             NAME_ERROR_MSG, name);
    }
    goto error;
}
Py_INCREF(v);
}
else {
    /* Slow-path if globals or builtins is not a dict */
    /* namespace 1: globals */
    v = PyObject_GetItem(f->f_globals, name);
    if (v == NULL) {
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;
        PyErr_Clear();
    }
    /* namespace 2: builtins */
    v = PyObject_GetItem(f->f_builtins, name);
    if (v == NULL) {
        if (PyErr_ExceptionMatches(PyExc_KeyError))
            format_exc_check_arg(
                PyExc_NameError,
                NAME_ERROR_MSG, name);
        goto error;
    }
}
}

```

The look-up algorithm for the `LOAD_GLOBAL` opcode first attempts to load the name from the `f_globals` and `f_builtins` fields if they are `dict` objects otherwise it attempts to fetch the value associated with the name from the `f_globals` or `f_builtins` object with the assumption that they implement some protocol for fetching the value associated with a given name. The value, if found, is loaded on to the evaluation stack using the `PUSH(v)` instruction otherwise, an error is set, and the execution jumps to the label for handling that error code.

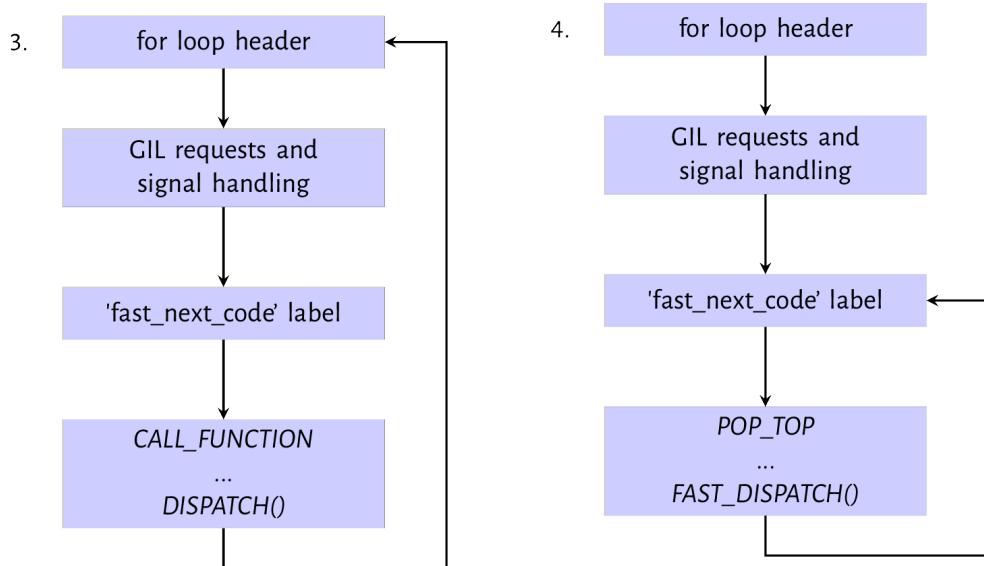
As the flow chart shows, the `DISPATCH()` macro, an alias for the `continue` statement, is called after the value is pushed onto the evaluation stack.

The second diagram, labelled 2 in figure 9.1, shows the execution of the `LOAD_CONST`. Listing 9.9 is an implementation of the `LOAD_CONST` opcode.

Listing 9.9: `LOAD_CONST` opcode implementation

```
PyObject *value = GETITEM(consts, oparg);
Py_INCREF(value);
PUSH(value);
FAST_DISPATCH();
```

This goes through the standard setup as `LOAD_GLOBAL` but after execution, `FAST_DISPATCH()` is called rather than `DISPATCH()`. This causes a jump to the `fast_next_opcode` code label from where the loop execution continues skipping the signal and GIL checks on the next iteration. Opcodes that have implementations that make C function calls make of the `DISPATCH` macro while opcodes like the `LOAD_GLOBAL` that does not make C function calls in their implementation make use of the `FAST_DISPATCH` macro. This means that the thread of execution can only yield the GIL after executing opcodes that make C function calls.

Figure 9.2: Evaluation path for `CALL_FUNCTION` and `POP_TOP` instruction

The next instruction is the `CALL_FUNCTION` opcode, as shown in the first image from figure 9.2. The compile emits this opcode for a function call with only positional arguments. Listing 9.10 is the implementation for this opcode. At the heart of the opcode implementation is the `call_function(&sp, oparg, NULL)`. `oparg` is the number of arguments passed to the function, and the `call_function` function reads that number of values from the evaluation stack.

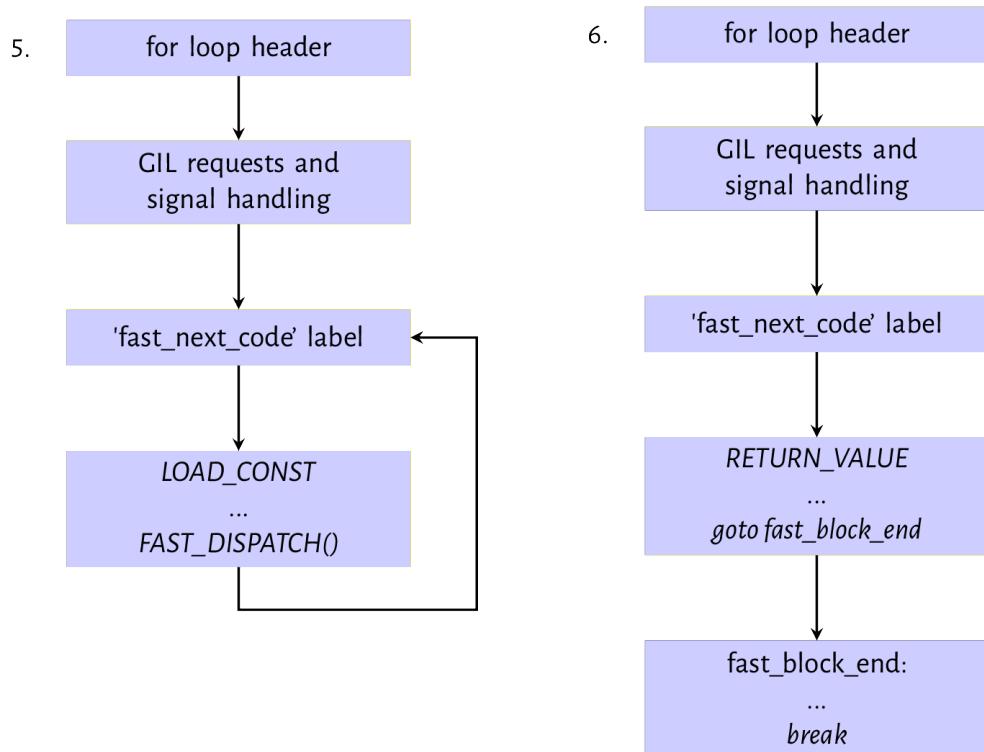
Listing 9.10: `CALL_FUNCTION` opcode implementation

```

PyObject **sp, *res;
PCALL(PCALL_ALL);
sp = stack_pointer;
res = call_function(&sp, oparg, NULL);
stack_pointer = sp;
PUSH(res);
if (res == NULL) {
    goto error;
}
DISPATCH();

```

The next instruction shown in diagram 4 of figure 9.2 is the `POP_TOP` instruction that removes a single value from the top of the evaluation stack - this clears any value placed on the stack by the previous function call.

Figure 9.3: Evaluation path for `LOAD_CONST` and `RETURN_VALUE` instruction

The next set of instructions are the `LOAD_CONST` and `RETURN_VALUE` pair shown in diagrams 5 and 6 of figure 9.3. The `LOAD_CONST` opcode loads a `None` value onto the evaluation stack; this is the value that the `RETURN_VALUE` will return. These two always go together when a function does not explicitly return any value. We have already looked at the mechanics of the `LOAD_CONST` instruction. The `RETURN_VALUE` instruction pops the top of the stack into the `retval` variable, sets the `WHY` status

code to `WHY_RETURN` and then performs a jump to the `fast_block_end` code label where the execution continues. If there has been no exception, the execution breaks out of the `for` loop and returns the `retval`.

Notice that a lot of the code snippets that we have looked at have the `goto error` jump, but we have intentionally discussing errors and exceptions out so far. We will look at exception handling in the next chapter. Although the function's bytecode looked at in this section is relatively trivial, it encapsulates the vanilla behaviour of the evaluation loop. Other opcodes may have more complicated implementations, but the essence of the execution is the same as described above.

Next, we look at some other interesting opcodes supported by the python virtual machine.

9.4 A sampling of opcodes

The python virtual machine has about 157 opcodes, so we randomly pick a few opcodes and deconstruct to get more of a feel for how these opcodes function. Some examples of these opcodes include:

1. `MAKE_FUNCTION`: As the name suggests, the opcode creates a function object from values on the evaluation stack. Consider a module containing the functions shown in listing 9.11.

Listing 9.11: Function definitions in a module

```
def test_non_local(arg, *args, defarg="test", defkwd=2, **kwd):
    local_arg = 2
    print(arg)
    print(defarg)
    print(args)
    print(defkwd)
    print(kwd)

def hello_world():
    print("Hello world!")
```

Disassembly of the code object from the module's compilation gives the set of bytecode instructions shown in listing 9.12

Listing 9.11: Disassembly of code object from listing 9.11

17	0 LOAD_CONST	8 (('test',))
	2 LOAD_CONST	1 (2)
	4 LOAD_CONST	2 (('defkwd',))
	6 BUILD_CONST_KEY_MAP	1
	8 LOAD_CONST	3 (<code object test_non_local at 0x109eed0\
0, file "string", line 17>)		
	10 LOAD_CONST	4 ('test_non_local')
	12 MAKE_FUNCTION	3
	14 STORE_NAME	0 (test_non_local)
45	16 LOAD_CONST	5 (<code object hello_world at 0x109eeae80,\
file "string", line 45>)		
	18 LOAD_CONST	6 ('hello_world')
	20 MAKE_FUNCTION	0
	22 STORE_NAME	1 (hello_world)
	24 LOAD_CONST	7 (None)
	26 RETURN_VALUE	

We can see that the `MAKE_FUNCTION` opcode appears twice in the series of bytecode instructions - one for each function definition within the module. The implementation of the `MAKE_FUNCTION` creates a function object and then stores the function in the local namespace using the function definition name. Notice that default arguments are pushed on the stack when such arguments are defined. The `MAKE_FUNCTION` implementation consumes these values by *and'ing* the oparg with a bitmask and popping values from the stack accordingly.

Listing 9.12: `MAKE_FUNCTION` opcode implementation

```

TARGET(MAKE_FUNCTION) {
    PyObject *qualname = POP();
    PyObject *codeobj = POP();
    PyFunctionObject *func = (PyFunctionObject *)
        PyFunction_NewWithQualName(codeobj, f->f_globals, qualname);

    Py_DECREF(codeobj);
    Py_DECREF(qualname);
    if (func == NULL) {
        goto error;
    }

    if (oparg & 0x08) {
        assert(PyTuple_CheckExact(TOP()));
        func->func_closure = POP();
    }
}

```

```

    if (oparg & 0x04) {
        assert(PyDict_CheckExact(TOP()));
        func->func_annotations = POP();
    }
    if (oparg & 0x02) {
        assert(PyDict_CheckExact(TOP()));
        func->func_kwdefaults = POP();
    }
    if (oparg & 0x01) {
        assert PyTuple_CheckExact(TOP());
        func->func_defaults = POP();
    }

    PUSH((PyObject *)func);
    DISPATCH();
}

```

The flags above denote the following.

1. `0x01`: a tuple of default argument objects in positional order is on the stack.
2. `0x02`: a dictionary of keyword-only parameters default values is on the stack.
3. `0x04`: an annotation dictionary is on the stack.
4. `0x08`: a tuple containing cells for free variables, making a closure is on the stack.

The `PyFunction_NewWithQualName` function that actually creates a function object is implemented in the `Objects/funcobject.c` module and its implementation is pretty simple. The function initializes a function object and sets values on the function object.

2. `LOAD_ATTR`: This opcode handles attribute references such as `x.y`. Assuming we have an instance object `x`, an attribute reference such as `x.name` translates to the set of opcodes shown in listing 9.13.

Listing 9.13: Opcodes for an attribute reference

24	LOAD_NAME	1	(x)
26	LOAD_ATTR	2	(name)
28	POP_TOP		
30	LOAD_CONST	4	(None)
32	RETURN_VALUE		

The `LOAD_ATTR` opcode implementation is pretty simple and shown in listing 9.14.

Listing 9.14: LOAD_ATTR opcode implementation

```

TARGET(LOAD_ATTR) {
    PyObject *name = GETITEM(names, oparg);
    PyObject *owner = TOP();
    PyObject *res = PyObject_GetAttr(owner, name);
    Py_DECREF(owner);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}

```

We have looked at the `PyObject_GetAttr` function in the chapter on objects. This function returns the value of an object's attribute which is then loaded on to the stack. One can review the chapter on objects to get the details on how this function works.

3. `CALL_FUNCTION_KW`: This opcode very similar in functionality to the `CALL_FUNCTION` opcode that was discussed previously but is used for function calls with keyword arguments. Listing 9.15 is the implementation for this opcode. Notice how one of the significant change from the implementation of the `CALL_FUNCTION` opcode is that a tuple of names is now passed as one of the arguments when `call_function` is invoked.

Listing 9.15: CALL_FUNCTION_KW opcode implementation

```

PyObject **sp, *res, *names;

names = POP();
assert(PyTuple_CheckExact(names) && PyTuple_GET_SIZE(names) <= oparg);
PCALL(PCALL_ALL);
sp = stack_pointer;
res = call_function(&sp, oparg, names);
stack_pointer = sp;
PUSH(res);
Py_DECREF(names);

if (res == NULL) {
    goto error;
}

```

The names are the keyword arguments of the function call, and they are used in the `_PyEval_EvalCodeWithName` to handle the setup before the code object for the function is executed.

This caps our explanation of the evaluation loop. As we have seen, the concepts behind the evaluation loop are not complicated - opcodes each have implementations that in C. These implementations are the actual do work functions. Two critical areas that we have not touched are exception handling and the block stack, two intimately related concepts that we look at in the following chapter.

10. The Block Stack

One of the data structures that does not get as much coverage as it should is the block stack, the other stack within a frame object. Most discussions of the Python VM mention the block stack passingly but then focus on the evaluation stack. However, the block stack is critical for exception handling. There are probably other methods of implementing exception handling but as will become evident as we progress through this chapter, using a stack, the block stack, makes it incredibly simple to implement exception handling. The block stack and exception handling are so intertwined that one will not fully understand the need for the block stack without actually considering exception handling. Loops also make use of the block stack, but it is difficult to see a reason for block stacks with loops until one looks at how loop constructs like `break` interact with exception handlers. The block stack makes the implementation of such interactions a straightforward affair.

The block stack is a stack data structure field within a frame object. Just like the evaluation stack, values are pushed to and popped from the block stack during the execution of a frame's code. However, the block stack is used only for handling loops and exceptions. The best way to explain the block stack is with an example, so we illustrate with a simple `try...finally` construct within a loop as shown in the snippet in listing 10.0.

Listing 10.0: Simple python function with exception handling

```
def test():
    for i in range(4):
        try:
            break
        finally:
            print("Exiting loop")
```

Listing 10.1 is the disassembly of code from Listing 10.0.

Listing 10.1: Disassembly of function in listing 10.0

2	0 SETUP_LOOP	34 (to 36)
	2 LOAD_GLOBAL	0 (range)
	4 LOAD_CONST	1 (4)
	6 CALL_FUNCTION	1
	8 GET_ITER	
>>	10 FOR_ITER	22 (to 34)
	12 STORE_FAST	0 (i)
3	14 SETUP_FINALLY	6 (to 22)

```

4           16 BREAK_LOOP
           18 POP_BLOCK
           20 LOAD_CONST      0 (None)

6    >>  22 LOAD_GLOBAL      1 (print)
           24 LOAD_CONST      2 ('Exiting loop')
           26 CALL_FUNCTION    1
           28 POP_TOP
           30 END_FINALLY
           32 JUMP_ABSOLUTE    10
>>  34 POP_BLOCK
>>  36 LOAD_CONST      0 (None)
           38 RETURN_VALUE

```

The combination of a `for` loop and `try ... finally` leads to a lot of instructions even for such a simple function as listing 10.1 shows. The opcodes of interest here are the `SETUP_LOOP` and `SETUP_FINALLY` opcodes so we look at the implementations of these to get the gist of what it does (all `SETUP_*` opcodes map to the same implementation).

The implementation for the `SETUP_LOOP` opcode is a simple function call - `PyFrame_BlockSetup(f, opcode, INSTR_OFFSET() + oparg, STACK_LEVEL())`. The arguments are pretty self-explanatory - `f` is the frame, `opcode` is the currently executing opcode, `INSTR_OFFSET() + oparg` is the instruction delta for the next instruction after that block (for the above code the delta is 50 for the `SETUP_LOOP`), and the `STACK_LEVEL` denotes how many items are on the value stack of the frame. The function call creates a new block and pushes it on the block stack. The information contained in this block is enough for the virtual machine to continue execution should something happen while in that block. Listing 10.2 is the implementation of this function.

Listing 10.2: Block setup code

```

void PyFrame_BlockSetup(PyFrameObject *f, int type, int handler, int level){
    PyTryBlock *b;
    if (f->f_iblock >= CO_MAXBLOCKS)
        Py_FatalError("XXX block stack overflow");
    b = &f->f_blockstack[f->f_iblock++];
    b->b_type = type;
    b->b_level = level;
    b->b_handler = handler;
}

```

The `handler` in Listing 10.2 is the pointer to the next instruction that should be executed after the `SETUP_*` block. A visual representation of the example from above is illustrative - figure 10.0 shows this using a subset of the bytecode.

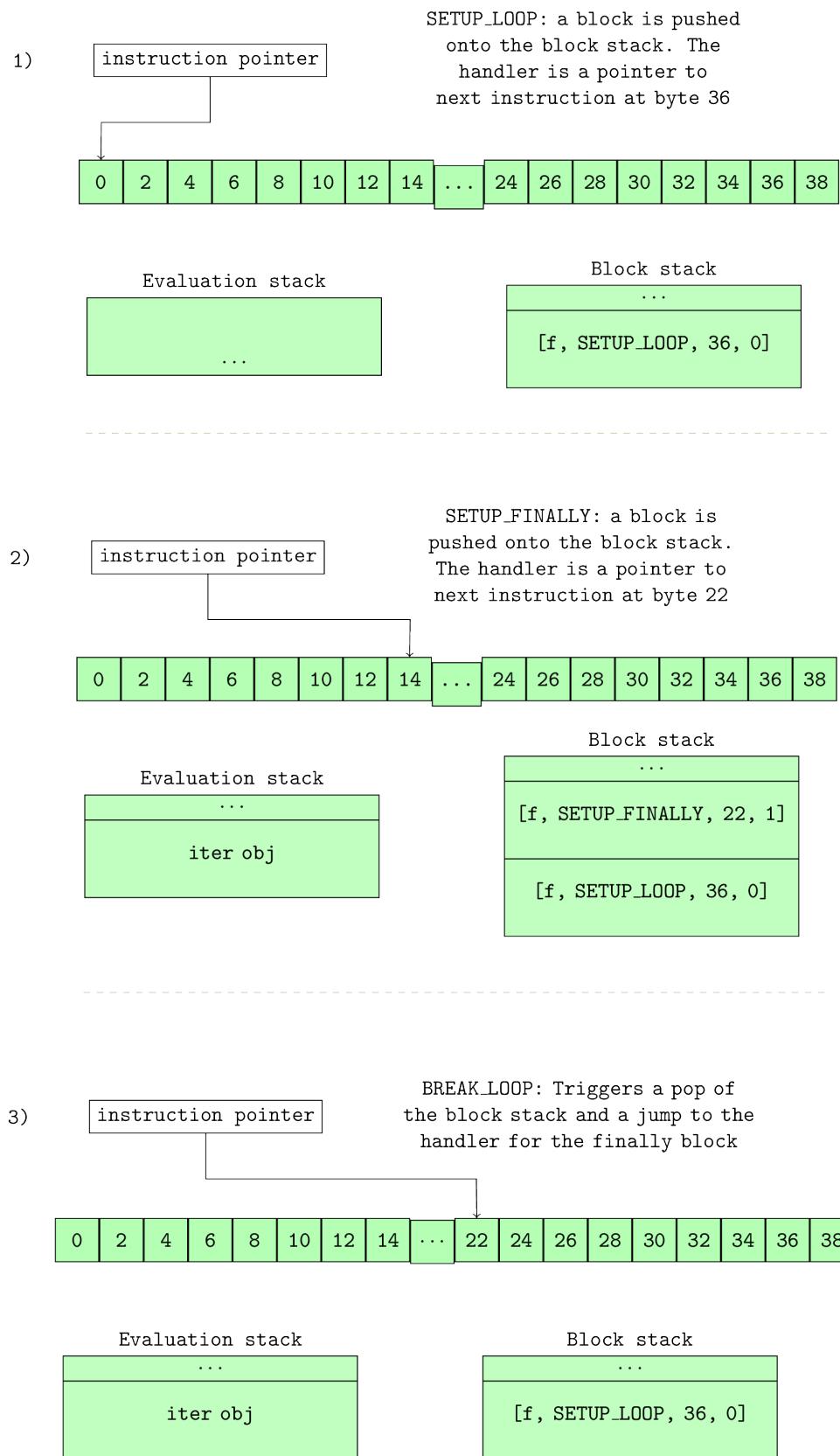
Figure 10.0: How the block stack changes with `SETUP_*` instructions

Figure 10.0 shows how the block stack varies during the execution of instructions.

The first diagram of figure 10.0 shows a single `SETUP_LOOP` block placed on the block stack after the execution of the `SETUP_LOOP` opcode. The instruction at offset 36 is the handler for this block, so when this stack is popped during normal execution, the interpreter will jump to that offset and continue execution. Another block is pushed onto the block stack during the execution of the `SETUP_FINALLY` opcode. We can see that as the stack is *Last In First Out* data structure, the `finally` block will be the first out - recall that `finally` must be executed regardless of the `break` statement.

A simple illustration of the use of a block stack is when the `break` statement is encountered while inside the exception handler within the loop. The `why` variable is set to `WHY_BREAK` during the execution of the `BREAK_LOOP` and a jump to the `fast_block_end` code label, where the block stack unwinding is handled, is performed.

The second diagram of figure 10.0 shows this. Unwinding is just a fancy name for popping blocks on the stack and executing their handler. So in this case, the `SETUP_FINALLY` block is popped off the stack, and the interpreter jumps to its handler at bytecode offset 22. Execution continues from that offset till the `END_FINALLY` statement. Since the `why` code is a `WHY_BREAK`; a jump is executed to the `fast_block_end` code label once again where more stack unwinding happens - the loop block is left on the stack. This time around (not shown in figure 10.0), the block popped from the stack has a handler at byte offset 36, so the execution continues at that bytecode offset, thus exiting the loop.

The use of the block stack dramatically simplifies the implementation of the virtual machine implementation. If loops are not implemented with a block stack, an opcode such as the `BREAK_LOOP` would need a jump destination. If we then throw in a `try..finally` construct with that `break` statement we get a complex implementation where we would have to keep track of optional jump destinations within `finally` blocks and so on.

10.1 A Short Note on Exception Handling

With this basic understanding of the block stack, it is not difficult to fathom the implementation of exceptions and exception handling. Take the snippet in Listing 10.3 that tries to add a number to a string.

Listing 10.3: Simple python function with exception handling

```
def test1():
    try:
        2 + 's'
    except Exception:
        print("Caught exception")
```

Listing 10.4 shows the opcodes generated from this simple function.

Listing 10.4: Disassembly of function in listing 10.3

2	0 SETUP_EXCEPT	12 (to 14)
3	2 LOAD_CONST	1 (2)
	4 LOAD_CONST	2 ('s')
	6 BINARY_ADD	
	8 POP_TOP	
	10 POP_BLOCK	
	12 JUMP_FORWARD	28 (to 42)
4	>> 14 DUP_TOP	
	16 LOAD_GLOBAL	0 (Exception)
	18 COMPARE_OP	10 (exception match)
	20 POP_JUMP_IF_FALSE	40
	22 POP_TOP	
	24 POP_TOP	
	26 POP_TOP	
5	28 LOAD_GLOBAL	1 (print)
	30 LOAD_CONST	3 ('Caught exception')
	32 CALL_FUNCTION	1
	34 POP_TOP	
	36 POP_EXCEPT	
	38 JUMP_FORWARD	2 (to 42)
>>	40 END_FINALLY	
>>	42 LOAD_CONST	0 (None)
	44 RETURN_VALUE	

We should have a conceptual idea of how this code block will execute if an exception should occur given the previous explanation. In summary, we expect the `PyNumber_Add` function from the `Objects/abstract.c` module to return a `NULL` for the `BINARY_ADD` opcode. In addition to returning a `NULL` value, the function also sets exception values on the currently executing thread's thread state. Recall, that the thread state has the `curexc_type`, `curexc_value` and `curexc_traceback` fields for holding the current exception in the thread of execution; these fields prove very useful while unwinding the block stack in search of exception handlers. You can follow the chain of function calls from the `binop_type_error` function in the `Objects/abstract.c` module all the way to the `PyErr_SetObject` and `PyErr_Restore` functions in the `Python/errors.c` module where the exception gets set on the thread state.

With the exception values set on the currently executing thread's thread state and a `NULL` value returned from the function call, the interpreter loop executes a jump to the error label where all the magic of exception handling occurs or not. For our trivial example above, we have only one block on the block stack, the `SETUP_EXCEPT` block with a handler at bytecode offset 14. The stack unwinding

begins after the jump to error handler label. The exception values - traceback, exception value and exception type, are pushed on top of the value stack, the SETUP_EXCEPT handler gets popped from the block stack, and then there is a jump to the handler - byte offset 14 in this case where the execution continues. Observe the bytecodes from offset 16 to offset 20 in listing 10.4; here the `Exception` class is loaded onto the stack, and then compared with the raised exception value present on the stack. If the exceptions match then normal execution can continue with the popping of exception and tracebacks from value stack and execution of any of the error handler code. When there is no exception match, the `END_FINALLY` instruction is executed, and since there is still an exception on the stack, there is a break from the exception loop.

In the case where there is no exception handling mechanism, the opcodes for the `test` function are more straightforward as shown in listing 10.5.

Listing 10.5: Disassembly of function in listing 10.3 when there is no exception handling

2	0 LOAD_CONST	1 (2)
	2 LOAD_CONST	2 ('s')
	4 BINARY_ADD	
	6 POP_TOP	
	8 LOAD_CONST	0 (None)
10	RETURN_VALUE	

The opcodes do not place any value on the block stack so when an exception followed by a jump to the error handling label occurs, there is no block to be unwound from the stack causing the loop to be exit and the error to be dumped to the user.

Although some implementation details are left out, this chapter covers the fundamentals of the interaction between the block stack and error handling in the Python virtual machine. There are other details such as handling exceptions within exceptions, nested exception handlers and so on. However, we conclude this short intermezzo at this point.

11. From Class code to bytecode

We have covered a lot of ground discussing the nuts and bolts of how the Python virtual machine or interpreter (whichever you want to call it) executes your code but for an object-oriented language like Python, we have left out one of the essential parts - the nuts and bolts of how a user-defined class gets compiled down to bytecode and executed.

Our discussions on Python objects have provided us with a rough idea of how new classes *may* be created; however, this intuition may not fully capture the whole process from the moment a user defines a `class` in source to the bytecode resulting from compiling that class. This chapter aims to bridge that gap and provide an exposition on this process.

As usual, we start with the straightforward user-defined class listing 11.0.

Listing 11.0: A simple class definition

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Listing 11.1 is the disassembly of the class from Listing 11.0.

Listing 11.1: A simple class definition

```
0 LOAD_BUILD_CLASS
 2 LOAD_CONST          0 (<code object Person at 0x102298b70, file "str\
ing", line 2>)
 4 LOAD_CONST          1 ('Person')
 6 MAKE_FUNCTION        0
 8 LOAD_CONST          1 ('Person')
10 CALL_FUNCTION        2
12 STORE_NAME           0 (Person)
14 LOAD_CONST          2 (None)
16 RETURN_VALUE
```

We are interested in bytes 0 to 12, the actual opcodes that create the new class object and store it for future reference by its name (`Person` in our example). Before, we expand on the opcodes above; we look at the process of class creation as specified by the [Python documentation¹](https://docs.python.org/3.6/reference/datamodel.html#customizing-class-creation). The description of

¹<https://docs.python.org/3.6/reference/datamodel.html#customizing-class-creation>

the process in the documentation, though done at a very high level, is pretty straightforward. We infer from [Python documentation](#)² that the class creation process involves the following steps.

1. The body of the class statement is isolated into a code object.
2. The appropriate metaclass for class instantiation is determined.
3. A class dictionary representing the namespace for the class is prepared.
4. The code object representing the class' body is executed within this namespace.
5. The class object is created.

During the final step, the class object is created by instantiating the `type` class, passing in the class name, base classes and class dictionary as arguments. Any `__prepare__` hooks are run before instantiating the class object. The metaclass used in the class object creation can be explicitly specified by supplying the `metaclass` keyword argument in the `class` definition. If a `metaclass` is not provided, the interpreter examines the first entry in the *tuple* of any base classes. If base classes are not used, the interpreter searches for the global variable `__metaclass__` and if not found, Python uses the default meta-class.

The whole class creation process starts with a load of the `__build_class__` function onto the value stack. This function is responsible for all the type creation heavy lifting. Next, type's body code object, already compiled into a code object, is loaded on the stack by the instruction at offset 2 - `LOAD_CONST`. This code object is then wrapped into a function object by the `MAKE_FUNCTION` opcode and placed back on the stack; it will soon become clear why this happens. By offset 10; the evaluation stack looks similar to that in Figure 11.0.

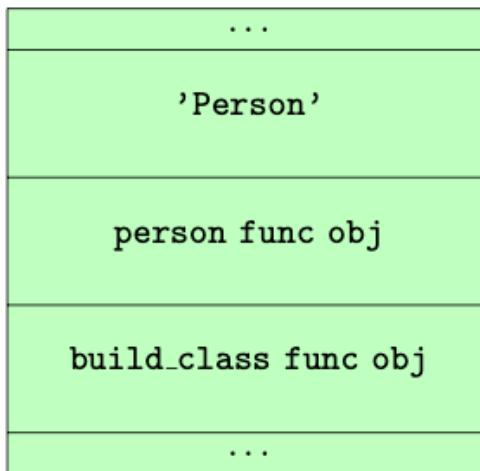


Figure 11.0: State of evaluation stack just before `CALL_FUNCTION`

At offset 10, `CALL_FUNCTION` handles invokes the `__build_class__` function with the two values above it on the evaluation stack as argument (the argument to `CALL_FUNCTION` is two). The `__build_class__` function in the `Python/bltinmodule.c` module is the workhorse that creates our class. A significant part of the function is devoted to sanity checks - check for the right arguments,

²<https://docs.python.org/3.6/reference/datamodel.html#customizing-class-creation>

checks for correct type etc. After these sanity checks, the function then has to decide on the right metaclass. The rules for determining the correct `metaclass` are reproduced verbatim from the [Python documentation](#)³.

1. if no bases and no explicit metaclass are given, then `type()` is used
2. if an explicit metaclass is given and it is not an instance of `type()`, then it is used directly as the metaclass
3. if an instance of `type()` is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used

The most derived metaclass is selected from the explicitly specified metaclass (if any) and the metaclasses (i.e. `type(c1s)`) of all specified base classes. The most derived metaclass is one which is a subtype of all of these candidate metaclasses. If none of the candidate metaclasses meets that criterion, then the class definition will fail with `TypeError`.

The actual snippet from the `__build_class` function that handles the metaclass resolution is in listing 11.2, and it has been annotated a bit more to provide some more clarity.

Listing 11.2: A simple class definition

```

...
/* kwds are values passed into brackets that follow class name
e.g class(metaclass=blah)*/
if (kwds == NULL) {
    meta = NULL;
    mkw = NULL;
}
else {
    mkw = PyDict_Copy(kwds); /* Don't modify kwds passed in! */
    if (mkw == NULL) {
        Py_DECREF(bases);
        return NULL;
    }
    /* for all intent and purposes &PyId_metaclass references the string "me\
taclass"
    but the &PyId_* macro handles static allocation of such strings */

    meta = _PyDict_GetItemId(mkw, &PyId_metaclass);
    if (meta != NULL) {
        Py_INCREF(meta);
        if (_PyDict_DelItemId(mkw, &PyId_metaclass) < 0) {
            Py_DECREF(meta);
            Py_DECREF(mkw);
        }
    }
}

```

³<https://docs.python.org/3.5/reference/datamodel.html#determining-the-appropriate-metaclass>

```

        Py_DECREF(bases);
        return NULL;
    }
    /* metaclass is explicitly given, check if it's indeed a class */
    isclass = PyType_Check(meta);
}
}
if (meta == NULL) {
    /* if there are no bases, use type: */
    if (PyTuple_GET_SIZE(bases) == 0) {
        meta = (PyObject *) (&PyType_Type);
    }
    /* else get the type of the first base */
    else {
        PyObject *base0 = PyTuple_GET_ITEM(bases, 0);
        meta = (PyObject *) (base0->ob_type);
    }
    Py_INCREF(meta);
    isclass = 1; /* meta is really a class */
}
...

```

With the metaclass found, `__build_class` then proceeds to check if any `__prepare__` attribute exists on the metaclass; if any such attribute exists the class namespace is *prepared* by executing the `__prepare__` hook passing the class name, class bases and any additional keyword arguments from the class definition. This hook is used to customize class behaviour. The following example in listing 11.3 which is taken from the example on metaclass definition and use of the [python documentation](#)⁴ shows an example of how the `__prepare__` hook can be used to implement a class with attribute ordering.

Listing 11.3: A simple meta-class definition

```

class OrderedClass(type):

    @classmethod
    def __prepare__(metacls, name, bases, **kwds):
        return collections.OrderedDict()

    def __new__(cls, name, bases, namespace, **kwds):
        result = type.__new__(cls, name, bases, dict(namespace))
        result.members = tuple(namespace)
        return result

```

⁴<https://docs.python.org/3.6/reference/datamodel.html#metaclass-example>

```

class A(metaclass=OrderedClass):
    def one(self): pass
    def two(self): pass
    def three(self): pass
    def four(self): pass

>>> A.members
('__module__', 'one', 'two', 'three', 'four')

```

The `__build_class` function returns an empty new dict if there is no `__prepare__` attribute defined on the metaclass but if there is one, the namespace that is used is the result of executing the `__prepare__` attribute like the snippet in listing 11.4.

Listing 11.4: Preparing for a new class

```

...
    // get the __prepare__ attribute
    prep = _PyObject_GetAttrId(meta, &PyId__prepare__);
    if (prep == NULL) {
        if (PyErr_ExceptionMatches(PyExc_AttributeError)) {
            PyErr_Clear();
            ns = PyDict_New(); // namespace is a new dict if __prepare__ is not \
defined
        }
        else {
            Py_DECREF(meta);
            Py_XDECREF(mkw);
            Py_DECREF(bases);
            return NULL;
        }
    }
    else {
        /** where __prepare__ is defined, the namespace is the result of executi\
ng
        the __prepare__ attribute ***/
        PyObject *pargs[2] = {name, bases};
        ns = _PyObject_FastCallDict(prep, pargs, 2, mkw);
        Py_DECREF(prep);
    }
    if (ns == NULL) {
        Py_DECREF(meta);
        Py_XDECREF(mkw);
        Py_DECREF(bases);
    }

```

```

    return NULL;
}
...

```

After handling the `__prepare__` hook, it is now time for the actual class object to be created. First, the execution of the class body's code object happens within the namespace from the previous paragraph. To understand why take a look at this code object's bytecode in listing 11.5.

Listing 11.5: Disassembly of code object for class body from listing 11.0

1	0 LOAD_NAME	0 (<code>__name__</code>)
	2 STORE_NAME	1 (<code>__module__</code>)
	4 LOAD_CONST	0 (<code>'test'</code>)
	6 STORE_NAME	2 (<code>__qualname__</code>)
2	8 LOAD_CONST	1 (<code object <code>__init__</code> at 0x102a80660, file "string", line 2>)
	10 LOAD_CONST	2 (<code>'test.__init__'</code>)
	12 MAKE_FUNCTION	0
	14 STORE_NAME	3 (<code>__init__</code>)
	16 LOAD_CONST	3 (<code>None</code>)
	18 RETURN_VALUE	

After executing this code object, the namespace will contain all the attributes of the class, i.e. class attributes, methods etc. This namespace subsequently used as an argument for a function call to the metaclass as shown in listing 11.6.

Listing 11.6: Invoking a metaclass to create a new class instance

```

// evaluate code object for body within namespace
none = PyEval_EvalCodeEx(PyFunction_GET_CODE(func), PyFunction_GET_GLOBALS(func),
), ns,
    NULL, 0, NULL, 0, NULL, 0, NULL,
    PyFunction_GET_CLOSURE(func));

if (none != NULL) {
    PyObject *margs[3] = {name, bases, ns};
    /**
     * this will 'call' the metaclass creating a new class object
     */
    cls = _PyObject_FastCallDict(meta, margs, 3, mkw);
    Py_DECREF(none);
}

```

Assuming we are using the `type` metaclass, calling the `type` means dereferencing the attribute in the `tp_call` slot of the class. The `tp_call` function then, in turn, dereferences the attribute in the `tp_new` slot which creates and returns our brand new class object. The `c1s` value returned is then placed back on the stack and stored to the `Person` variable. There we have it, the process of creating a new class and this is all there is to it in Python.

12. Generators: Behind the scenes.

Generators are one of the beautiful concepts in Python. A generator function is one that contains a `yield` statement, and when called, it returns a generator. A simple use of generators in Python is as an iterator that produces values for an iteration on demand. Listing 12.0 is a simple example of a generator function that produces values from 0 up to n .

Listing 12.0: A simple generator

```
def firstn(n):
    num = 0
    while num < n:
        v = yield num
        print(v)
        num += 1
```

`firstn` contains the `yield` statement, so calling it will not return a simple value as a conventional function would do. Instead, it will return a generator object which captures the **continuation** of the computation. We can then use the `next` function to get successive values from the returned generator object or send values into the generator using the `send` method of the generator object.

In this chapter, we are not interested in the semantics of the generators objects or the right way to use them. Our interest is in how generators are implemented under the covers in CPython. We are interested in how it is possible to suspend a computation and then subsequently resume such computation. We look at the data structures and ideas behind this concept, and surprisingly, they are not too complicated. First, we look at the C implementation of a generator object.

12.1 The Generator object

Listing 12.1 is the definition of a generator object, and going through this definition provides some intuition into how a generator execution can be suspended or resumed. We can see that a generator object contains a *frame* object and a *code* object, two objects that are essential to the execution of Python bytecode.

Listing 12.1: Generator object definition

```

/* _PyGenObject_HEAD defines the initial segment of generator
and coroutine objects. */
#define _PyGenObject_HEAD(prefix) \
    PyObject_HEAD \
    /* Note: gi_frame can be NULL if the generator is "finished" */ \
    struct _frame *prefix##_frame; \
    /* True if generator is being executed. */ \
    char prefix##_running; \
    /* The code object backing the generator */ \
    PyObject *prefix##_code; \
    /* List of weak reference. */ \
    PyObject *prefix##_weakreflist; \
    /* Name of the generator. */ \
    PyObject *prefix##_name; \
    /* Qualified name of the generator. */ \
    PyObject *prefix##_qualname;

typedef struct {
    /* The gi_ prefix is intended to remind of generator-iterator. */
    _PyGenObject_HEAD(gi)
} PyGenObject;

```

The following comprise the main attributes of a generator object.

1. `prefix##_frame`: This field references a frame object. This frame object contains the code object of a generator and it is within this frame that the execution of the generator object's code object takes place.
2. `prefix##_running`: This is a boolean field that indicates whether the generator is running.
3. `prefix##_code`: This field references the code object associated with the generator. This is the code object that executes whenever the generator is running.
4. `prefix##_name`: This is the name of the generator - in listing 12.0, the value is `firstrn`.
5. `prefix##_qualname`: This is the fully qualified name of the generator. Most times this value is the same as that of `prefix##_name`.

Creating generators

When we call a generator function, the generator function does not run to completion and return a value; instead, it produces a generator object. This is due to the `CO_GENERATOR` flag that gets set when compiling a generator function. This flag comes in very useful during the setup process that happens just before the code object execution.

During the execution of the code object for the function, recall the `_PyEval_EvalCodeWithName` is invoked to perform some setup. During this setup process, the interpreter checks if the `CO_GENERATOR` flag; if set, it creates and returns a generator object rather than call the evaluation loop function. The magic happens at the last code block of the `_PyEval_EvalCodeWithName` as shown in listing 12.2.

Listing 12.2: `_PyEval_EvalCodeWithName` returns a generator object when processing a code object with generator flag

```

/* Handle generator/coroutine/asynchronous generator */
if (co->co_flags & (CO_GENERATOR | CO_COROUTINE | CO_ASYNC_GENERATOR)) {
    PyObject *gen;
    PyObject *coro_wrapper = tstate->coroutine_wrapper;
    int is_coro = co->co_flags & CO_COROUTINE;

    if (is_coro && tstate->in_coroutine_wrapper) {
        assert(coro_wrapper != NULL);
        PyErr_Format(PyExc_RuntimeError,
                    "coroutine wrapper %.200R attempted "
                    "to recursively wrap %.200R",
                    coro_wrapper,
                    co);
        goto fail;
    }

/* Don't need to keep the reference to f_back; it will be set
* when the generator is resumed. */
Py_CLEAR(f->f_back);

PCALL(PCALL_GENERATOR);

/* Create a new generator that owns the ready to run frame
* and return that as the value. */
if (is_coro) {
    gen = PyCoro_New(f, name, qualname);
} else if (co->co_flags & CO_ASYNC_GENERATOR) {
    gen = PyAsyncGen_New(f, name, qualname);
} else {
    gen = PyGen_NewWithQualName(f, name, qualname);
}
if (gen == NULL)
    return NULL;

if (is_coro && coro_wrapper != NULL) {
    PyObject *wrapped;
    tstate->in_coroutine_wrapper = 1;
}

```

```

wrapped = PyObject_CallFunction(coro_wrapper, "N", gen);
tstate->in_coroutine_wrapper = 0;
return wrapped;
}

return gen;
}

```

We can see from Listing 12.2 that bytecode for a generator function code object is never executed at the point of the function call - the execution of bytecode only happens when the returned generator object is running, and we look at this next.

12.2 Running a generator

We can run a generator object by passing it as an argument to the `next` builtin function. This will cause the generator to execute until it hits a `yield` expression then it suspends execution. The critical question here is how the generators can capture the execution state and update those at will.

Looking back at the generator object definition in Listing 12.1, we see that generators have a field that references a frame object, and this gets filled when the generator is created as shown in listing 12.2. The frame object as we recall has all the state that is required to execute a code object so by having a reference to that execution frame, the generator object can capture all the state required for its execution.

Now that we know how a generator object captures execution state, we move to the question of how the execution of a suspended generator object is resumed, and this is not too hard to figure out given the information that we have already. When the `next` builtin function is called with a generator as an argument, the `next` function dereferences the `tp_iternext` field of the generator type and invokes whatever function that field references. In the case of a generator object, that field references a function, `gen_iternext`, which calls the `gen_send_ex` function, that does the actual work of resuming the execution of the generator object. Before the generator object was created, the initial setup of the frame object and variables was carried out by the `_PyEval_EvalCodeWithName` function, so the execution of the generator object involves calling the `PyEval_EvalFrameEx` with the frame object contained within the generator object as the frame argument. The execution of the code object contained within the frame then proceeds as explained the chapter on the evaluation loop.

To get a more in-depth look at a generator function, we look at the generator function in listing 12.0. The disassembly of the generator function in listing 12.0 results in the set of bytecode shown in listing 12.3.

Listing 12.3: Disassembly of generator function from listing 12.0

4	0 LOAD_CONST	1 (0)
	2 STORE_FAST	1 (num)
5	4 SETUP_LOOP	34 (to 40)
>>	6 LOAD_FAST	1 (num)
	8 LOAD_FAST	0 (n)
	10 COMPARE_OP	0 (<)
	12 POP_JUMP_IF_FALSE	38
6	14 LOAD_FAST	1 (num)
	16 YIELD_VALUE	
	18 STORE_FAST	2 (v)
7	20 LOAD_GLOBAL	0 (print)
	22 LOAD_FAST	2 (v)
	24 CALL_FUNCTION	1
	26 POP_TOP	
8	28 LOAD_FAST	1 (num)
	30 LOAD_CONST	2 (1)
	32 INPLACE_ADD	
	34 STORE_FAST	1 (num)
	36 JUMP_ABSOLUTE	6
>>	38 POP_BLOCK	
>>	40 LOAD_CONST	0 (None)
	42 RETURN_VALUE	

When the execution of the bytecode shown in listing 12.3 for the generator function gets to the YIELD_VALUE opcode at byte offset 16, that opcode causes the evaluation to suspend and return the value on the top of the stack to the caller. By suspend, we mean the evaluation loop for the currently executing frame is exited however this frame is not deallocated because it is still referenced by the generator object so the execution of the frame can continue again when PyEval_EvalFrameEx is invoked with the frame as one of its arguments.

Python generators do more than just **generate** values; they can also consume values by using the generator `send` method. This is possible because `yield` is an expression that evaluates to a value. When the `send` method is called on a generator with a value, the `gen_send_ex` method places the value onto the evaluation stack of the generator object frame before the evaluation of the frame object resumes. Listing 12.3 shows the `STORE_FAST` instruction comes after `YIELD_VALUE`; this stores the value at the top of the stack to the provided name. In the case where there is no `send` function call, then the `None` value is placed on the top of the stack.