Effective Software Development Series
Scott Meyers, Consulting Editor

Effective Python

59 Specific Ways to Write Better Python

Brett Slatkin

Praise for *Effective Python*

"Each item in Slatkin's *Effective Python* teaches a self-contained lesson with its own source code. This makes the book random-access: Items are easy to browse and study in whatever order the reader needs. I will be recommending *Effective Python* to students as an admirably compact source of mainstream advice on a very broad range of topics for the intermediate Python programmer."

—Brandon Rhodes, software engineer at Dropbox and chair of PyCon 2016-2017

"I've been programming in Python for years and thought I knew it pretty well. Thanks to this treasure trove of tips and techniques, I realize there's so much more I could be doing with my Python code to make it faster (e.g., using built-in data structures), easier to read (e.g., enforcing keyword-only arguments), and much more Pythonic (e.g., using zip to iterate over lists in parallel)."

—Pamela Fox, educationeer, Khan Academy

"If I had this book when I first switched from Java to Python, it would have saved me many months of repeated code rewrites, which happened each time I realized I was doing particular things 'non-Pythonically.' This book collects the vast majority of basic Python 'must-knows' into one place, eliminating the need to stumble upon them one-by-one over the course of months or years. The scope of the book is impressive, starting with the importance of PEP8 as well as that of major Python idioms, then reaching through function, method and class design, effective standard library use, quality API design, testing, and performance measurement—this book really has it all. A fantastic introduction to what it really means to be a Python programmer for both the novice and the experienced developer."

—Mike Bayer, creator of SQLAlchemy

"*Effective Python* will take your Python skills to the next level with clear guidelines for improving Python code style and function."

—Leah Culver, developer advocate, Dropbox

"This book is an exceptionally great resource for seasoned developers in other languages who are looking to quickly pick up Python and move beyond the basic language constructs into more Pythonic code. The organization of the book is clear, concise, and easy to digest, and each item and chapter can stand on its own as a meditation on a particular topic. The book covers the breadth of language constructs in pure Python without confusing the reader with the complexities of the broader Python ecosystem. For more seasoned developers the book provides in-depth examples of language constructs they may not have previously encountered, and provides examples of less commonly used language features. It is clear that the author is exceptionally facile with Python, and he uses his professional experience to alert the reader to common subtle bugs and common failure modes. Furthermore, the book does an excellent job of pointing out subtleties between Python 2.X and Python 3.X and could serve as a refresher course as one transitions between variants of Python."

—Katherine Scott, software lead, Tempo Automation

“This is a great book for both novice and experienced programmers. The code examples and explanations are well thought out and explained concisely and thoroughly.”

—C. Titus Brown, associate professor, UC Davis

“This is an immensely useful resource for advanced Python usage and building cleaner, more maintainable software. Anyone looking to take their Python skills to the next level would benefit from putting the book’s advice into practice.”

—Wes McKinney, creator of pandas; author of Python for Data Analysis*; and software engineer at Cloudera*

Effective Python

Effective Python

59 SPECIFIC WAYS TO WRITE BETTER PYTHON

Brett Slatkin

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Slatkin, Brett, author.
Effective Python : 59 specific ways to write better Python / Brett Slatkin.
pages cm
Includes index.
ISBN 978-0-13-403428-7 (pbk. : alk. paper)—ISBN 0-13-403428-7 (pbk. : alk. paper)
1. Python (Computer program language) 2. Computer programming. I. Title.
QA76.73.P98S57 2015
005.13'3—dc23

2014048305

ISBN-13: 978-0-13-403428-7
ISBN-10: 0-13-403428-7

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, March 2015

Editor-in-Chief
Mark L. Taub

Senior Acquisitions Editor
Trina MacDonald

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Copy Editor
Stephanie Geels

Indexer
Jack Lewis

Proofreader
Melissa Panagos

Technical Reviewers
Brett Cannon
Tavis Rudd
Mike Taylor

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
LaurelTech

To our family, loved and lost

This page intentionally left blank

Contents

Preface **xiii**

Acknowledgments **xvii**

About the Author **xix**

Chapter 1: Pythonic Thinking **1**

Item 1: Know Which Version of Python You're Using 1
Item 2: Follow the PEP 8 Style Guide 2
Item 3: Know the Differences Between `bytes`, `str`, and `unicode` 5
Item 4: Write Helper Functions Instead of Complex Expressions 8
Item 5: Know How to Slice Sequences 10
Item 6: Avoid Using `start`, `end`, and `stride` in a Single Slice 13
Item 7: Use List Comprehensions Instead of `map` and `filter` 15
Item 8: Avoid More Than Two Expressions in List Comprehensions 16
Item 9: Consider Generator Expressions for Large Comprehensions 18
Item 10: Prefer `enumerate` Over `range` 20
Item 11: Use `zip` to Process Iterators in Parallel 21
Item 12: Avoid `else` Blocks After `for` and `while` Loops 23
Item 13: Take Advantage of Each Block in `try`/`except`/`else`/`finally` 26

Chapter 2: Functions **29**

Item 14: Prefer Exceptions to Returning `None` 29
Item 15: Know How Closures Interact with Variable Scope 31

Item 16: Consider Generators Instead of Returning Lists 36
Item 17: Be Defensive When Iterating Over Arguments 38
Item 18: Reduce Visual Noise with Variable Positional Arguments 43
Item 19: Provide Optional Behavior with Keyword Arguments 45
Item 20: Use `None` and Docstrings to Specify Dynamic Default Arguments 48
Item 21: Enforce Clarity with Keyword-Only Arguments 51

Chapter 3: Classes and Inheritance 55

Item 22: Prefer Helper Classes Over Bookkeeping with Dictionaries and Tuples 55
Item 23: Accept Functions for Simple Interfaces Instead of Classes 61
Item 24: Use `@classmethod` Polymorphism to Construct Objects Generically 64
Item 25: Initialize Parent Classes with `super` 69
Item 26: Use Multiple Inheritance Only for Mix-in Utility Classes 73
Item 27: Prefer Public Attributes Over Private Ones 78
Item 28: Inherit from `collections.abc` for Custom Container Types 83

Chapter 4: Metaclasses and Attributes 87

Item 29: Use Plain Attributes Instead of Get and Set Methods 87
Item 30: Consider `@property` Instead of Refactoring Attributes 91
Item 31: Use Descriptors for Reusable `@property` Methods 95
Item 32: Use `__getattr__`, `__getattribute__`, and `__setattr__` for Lazy Attributes 100
Item 33: Validate Subclasses with Metaclasses 105
Item 34: Register Class Existence with Metaclasses 108
Item 35: Annotate Class Attributes with Metaclasses 112

Chapter 5: Concurrency and Parallelism 117

Item 36: Use `subprocess` to Manage Child Processes 118
Item 37: Use Threads for Blocking I/O, Avoid for Parallelism 122
Item 38: Use `Lock` to Prevent Data Races in Threads 126
Item 39: Use `Queue` to Coordinate Work Between Threads 129

Item 40: Consider Coroutines to Run Many Functions Concurrently 136
Item 41: Consider concurrent.futures for True Parallelism 145

Chapter 6: Built-in Modules 151
Item 42: Define Function Decorators with functools.wraps 151
Item 43: Consider contextlib and with Statements for Reusable try/finally Behavior 153
Item 44: Make pickle Reliable with copyreg 157
Item 45: Use datetime Instead of time for Local Clocks 162
Item 46: Use Built-in Algorithms and Data Structures 166
Item 47: Use decimal When Precision Is Paramount 171
Item 48: Know Where to Find Community-Built Modules 173

Chapter 7: Collaboration 175
Item 49: Write Docstrings for Every Function, Class, and Module 175
Item 50: Use Packages to Organize Modules and Provide Stable APIs 179
Item 51: Define a Root Exception to Insulate Callers from APIs 184
Item 52: Know How to Break Circular Dependencies 187
Item 53: Use Virtual Environments for Isolated and Reproducible Dependencies 192

Chapter 8: Production 199
Item 54: Consider Module-Scoped Code to Configure Deployment Environments 199
Item 55: Use repr Strings for Debugging Output 202
Item 56: Test Everything with unittest 204
Item 57: Consider Interactive Debugging with pdb 208
Item 58: Profile Before Optimizing 209
Item 59: Use tracemalloc to Understand Memory Usage and Leaks 214

Index 217

This page intentionally left blank

Preface

The Python programming language has unique strengths and charms that can be hard to grasp. Many programmers familiar with other languages often approach Python from a limited mindset instead of embracing its full expressivity. Some programmers go too far in the other direction, overusing Python features that can cause big problems later.

This book provides insight into the *Pythonic* way of writing programs: the best way to use Python. It builds on a fundamental understanding of the language that I assume you already have. Novice programmers will learn the best practices of Python's capabilities. Experienced programmers will learn how to embrace the strangeness of a new tool with confidence.

My goal is to prepare you to make a big impact with Python.

What This Book Covers

Each chapter in this book contains a broad but related set of items. Feel free to jump between items and follow your interest. Each item contains concise and specific guidance explaining how you can write Python programs more effectively. Items include advice on what to do, what to avoid, how to strike the right balance, and why this is the best choice.

The items in this book are for Python 3 and Python 2 programmers alike (see Item 1: "Know Which Version of Python You're Using"). Programmers using alternative runtimes like Jython, IronPython, or PyPy should also find the majority of items to be applicable.

Chapter 1: Pythonic Thinking

The Python community has come to use the adjective *Pythonic* to describe code that follows a particular style. The idioms of Python have emerged over time through experience using the language and working with others. This chapter covers the best way to do the most common things in Python.

Chapter 2: Functions

Functions in Python have a variety of extra features that make a programmer's life easier. Some are similar to capabilities in other programming languages, but many are unique to Python. This chapter covers how to use functions to clarify intention, promote reuse, and reduce bugs.

Chapter 3: Classes and Inheritance

Python is an object-oriented language. Getting things done in Python often requires writing new classes and defining how they interact through their interfaces and hierarchies. This chapter covers how to use classes and inheritance to express your intended behaviors with objects.

Chapter 4: Metaclasses and Attributes

Metaclasses and dynamic attributes are powerful Python features. However, they also enable you to implement extremely bizarre and unexpected behaviors. This chapter covers the common idioms for using these mechanisms to ensure that you follow the *rule of least surprise*.

Chapter 5: Concurrency and Parallelism

Python makes it easy to write concurrent programs that do many different things seemingly at the same time. Python can also be used to do parallel work through system calls, subprocesses, and C-extensions. This chapter covers how to best utilize Python in these subtly different situations.

Chapter 6: Built-in Modules

Python is installed with many of the important modules that you'll need to write programs. These standard packages are so closely intertwined with idiomatic Python that they may as well be part of the language specification. This chapter covers the essential built-in modules.

Chapter 7: Collaboration

Collaborating on Python programs requires you to be deliberate about how you write your code. Even if you're working alone, you'll want to understand how to use modules written by others. This chapter covers the standard tools and best practices that enable people to work together on Python programs.

Chapter 8: Production

Python has facilities for adapting to multiple deployment environments. It also has built-in modules that aid in hardening your programs and making them bulletproof. This chapter covers how to use Python to debug, optimize, and test your programs to maximize quality and performance at runtime.

Conventions Used in This Book

Python code snippets in this book are in `monospace font` and have syntax highlighting. I take some artistic license with the Python style guide to make the code examples better fit the format of a book or to highlight the most important parts. When lines are long, I use ➥ characters to indicate that they wrap. I truncate snippets with ellipses comments (#. . .) to indicate regions where code exists that isn't essential for expressing the point. I've also left out embedded documentation to reduce the size of code examples. I strongly suggest that you don't do this in your projects; instead, you should follow the style guide (see Item 2: "Follow the PEP 8 Style Guide") and write documentation (see Item 49: "Write Docstrings for Every Function, Class, and Module").

Most code snippets in this book are accompanied by the corresponding output from running the code. When I say "output," I mean console or terminal output: what you see when running the Python program in an interactive interpreter. Output sections are in monospace font and are preceded by a >>> line (the Python interactive prompt). The idea is that you could type the code snippets into a Python shell and reproduce the expected output.

Finally, there are some other sections in monospace font that are not preceded by a >>> line. These represent the output of running programs besides the Python interpreter. These examples often begin with $ characters to indicate that I'm running programs from a command-line shell like Bash.

Where to Get the Code and Errata

It's useful to view some of the examples in this book as whole programs without interleaved prose. This also gives you a chance to tinker with the code yourself and understand why the program works as described. You can find the source code for all code snippets in this book on the book's website (http://www.effectivepython.com). Any errors found in the book will have corrections posted on the website.

Acknowledgments

This book would not have been possible without the guidance, support, and encouragement from many people in my life.

Thanks to Scott Meyers for the Effective Software Development series. I first read *Effective C++* when I was 15 years old and fell in love with the language. There's no doubt that Scott's books led to my academic experience and first job at Google. I'm thrilled to have had the opportunity to write this book.

Thanks to my core technical reviewers for the depth and thoroughness of their feedback: Brett Cannon, Tavis Rudd, and Mike Taylor. Thanks to Leah Culver and Adrian Holovaty for thinking this book would be a good idea. Thanks to my friends who patiently read earlier versions of this book: Michael Levine, Marzia Niccolai, Ade Oshineye, and Katrina Sostek. Thanks to my colleagues at Google for their review. Without all of your help, this book would have been inscrutable.

Thanks to everyone involved in making this book a reality. Thanks to my editor Trina MacDonald for kicking off the process and being supportive throughout. Thanks to the team who were instrumental: development editors Tom Cirtin and Chris Zahn, editorial assistant Olivia Basegio, marketing manager Stephane Nakib, copy editor Stephanie Geels, and production editor Julie Nahil.

Thanks to the wonderful Python programmers I've known and worked with: Anthony Baxter, Brett Cannon, Wesley Chun, Jeremy Hylton, Alex Martelli, Neal Norwitz, Guido van Rossum, Andy Smith, Greg Stein, and Ka-Ping Yee. I appreciate your tutelage and leadership. Python has an excellent community and I feel lucky to be a part of it.

Thanks to my teammates over the years for letting me be the worst player in the band. Thanks to Kevin Gibbs for helping me take risks. Thanks to Ken Ashcraft, Ryan Barrett, and Jon McAlister for showing me how it's done. Thanks to Brad Fitzpatrick for taking it to the next level. Thanks to Paul McDonald for co-founding our crazy project. Thanks to Jeremy Ginsberg and Jack Hebert for making it a reality.

Thanks to the inspiring programming teachers I've had: Ben Chelf, Vince Hugo, Russ Lewin, Jon Stemmle, Derek Thomson, and Daniel Wang. Without your instruction, I would never have pursued our craft or gained the perspective required to teach others.

Thanks to my mother for giving me a sense of purpose and encouraging me to become a programmer. Thanks to my brother, my grandparents, and the rest of my family and childhood friends for being role models as I grew up and found my passion.

Finally, thanks to my wife, Colleen, for her love, support, and laughter through the journey of life.

About the Author

Brett Slatkin is a senior staff software engineer at Google. He is the engineering lead and co-founder of Google Consumer Surveys. He formerly worked on Google App Engine's Python infrastructure. He is the co-creator of the PubSubHubbub protocol. Nine years ago he cut his teeth using Python to manage Google's enormous fleet of servers.

Outside of his day job, he works on open source tools and writes about software, bicycles, and other topics on his personal website (http://onebigfluke.com). He earned his B.S. in computer engineering from Columbia University in the City of New York. He lives in San Francisco.

This page intentionally left blank

2 Functions

The first organizational tool programmers use in Python is the *function*. As in other programming languages, functions enable you to break large programs into smaller, simpler pieces. They improve readability and make code more approachable. They allow for reuse and refactoring.

Functions in Python have a variety of extra features that make the programmer's life easier. Some are similar to capabilities in other programming languages, but many are unique to Python. These extras can make a function's purpose more obvious. They can eliminate noise and clarify the intention of callers. They can significantly reduce subtle bugs that are difficult to find.

Item 14: Prefer Exceptions to Returning `None`

When writing utility functions, there's a draw for Python programmers to give special meaning to the return value of `None`. It seems to makes sense in some cases. For example, say you want a helper function that divides one number by another. In the case of dividing by zero, returning `None` seems natural because the result is undefined.

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

Code using this function can interpret the return value accordingly.

```
result = divide(x, y)
if result is None:
    print('Invalid inputs')
```

What happens when the numerator is zero? That will cause the return value to also be zero (if the denominator is non-zero). This can cause problems when you evaluate the result in a condition like an `if` statement. You may accidentally look for any `False` equivalent value to indicate errors instead of only looking for `None` (see Item 4: "Write Helper Functions Instead of Complex Expressions" for a similar situation).

```
x, y = 0, 5
result = divide(x, y)
if not result:
    print('Invalid inputs')  # This is wrong!
```

This is a common mistake in Python code when `None` has special meaning. This is why returning `None` from a function is error prone. There are two ways to reduce the chance of such errors.

The first way is to split the return value into a two-tuple. The first part of the tuple indicates that the operation was a success or failure. The second part is the actual result that was computed.

```
def divide(a, b):
    try:
        return True, a / b
    except ZeroDivisionError:
        return False, None
```

Callers of this function have to unpack the tuple. That forces them to consider the status part of the tuple instead of just looking at the result of division.

```
success, result = divide(x, y)
if not success:
    print('Invalid inputs')
```

The problem is that callers can easily ignore the first part of the tuple (using the underscore variable name, a Python convention for unused variables). The resulting code doesn't look wrong at first glance. This is as bad as just returning `None`.

```
_, result = divide(x, y)
if not result:
    print('Invalid inputs')
```

The second, better way to reduce these errors is to never return `None` at all. Instead, raise an exception up to the caller and make them deal with it. Here, I turn a `ZeroDivisionError` into a `ValueError` to indicate to the caller that the input values are bad:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs') from e
```

Now the caller should handle the exception for the invalid input case (this behavior should be documented; see Item 49: “Write Docstrings for Every Function, Class, and Module”). The caller no longer requires a condition on the return value of the function. If the function didn’t raise an exception, then the return value must be good. The outcome of exception handling is clear.

```
x, y = 5, 2
try:
    result = divide(x, y)
except ValueError:
    print('Invalid inputs')
else:
    print('Result is %.1f' % result)
```

```
>>>
Result is 2.5
```

Things to Remember

- ✦ Functions that return `None` to indicate special meaning are error prone because `None` and other values (e.g., zero, the empty string) all evaluate to `False` in conditional expressions.
- ✦ Raise exceptions to indicate special situations instead of returning `None`. Expect the calling code to handle exceptions properly when they’re documented.

Item 15: Know How Closures Interact with Variable Scope

Say you want to sort a list of numbers but prioritize one group of numbers to come first. This pattern is useful when you’re rendering a user interface and want important messages or exceptional events to be displayed before everything else.

A common way to do this is to pass a helper function as the `key` argument to a list’s `sort` method. The helper’s return value will be used as the value for sorting each item in the list. The helper can check whether the given item is in the important group and can vary the sort key accordingly.

```
def sort_priority(values, group):
    def helper(x):
        if x in group:
            return (0, x)
        return (1, x)
    values.sort(key=helper)
```

This function works for simple inputs.

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
sort_priority(numbers, group)
print(numbers)
```

```
>>>
[2, 3, 5, 7, 1, 4, 6, 8]
```

There are three reasons why this function operates as expected:

- Python supports *closures*: functions that refer to variables from the scope in which they were defined. This is why the `helper` function is able to access the `group` argument to `sort_priority`.
- Functions are *first-class* objects in Python, meaning you can refer to them directly, assign them to variables, pass them as arguments to other functions, compare them in expressions and `if` statements, etc. This is how the `sort` method can accept a closure function as the `key` argument.
- Python has specific rules for comparing tuples. It first compares items in index zero, then index one, then index two, and so on. This is why the return value from the `helper` closure causes the sort order to have two distinct groups.

It'd be nice if this function returned whether higher-priority items were seen at all so the user interface code can act accordingly. Adding such behavior seems straightforward. There's already a closure function for deciding which group each number is in. Why not also use the closure to flip a flag when high-priority items are seen? Then the function can return the flag value after it's been modified by the closure.

Here, I try to do that in a seemingly obvious way:

```
def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            found = True  # Seems simple
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

I can run the function on the same inputs as before.

```
found = sort_priority2(numbers, group)
print('Found:', found)
print(numbers)
```

```
>>>
Found: False
[2, 3, 5, 7, 1, 4, 6, 8]
```

The sorted results are correct, but the `found` result is wrong. Items from `group` were definitely found in `numbers`, but the function returned `False`. How could this happen?

When you reference a variable in an expression, the Python interpreter will traverse the scope to resolve the reference in this order:

1. The current function's scope
2. Any enclosing scopes (like other containing functions)
3. The scope of the module that contains the code (also called the *global scope*)
4. The built-in scope (that contains functions like `len` and `str`)

If none of these places have a defined variable with the referenced name, then a `NameError` exception is raised.

Assigning a value to a variable works differently. If the variable is already defined in the current scope, then it will just take on the new value. If the variable doesn't exist in the current scope, then Python treats the assignment as a variable definition. The scope of the newly defined variable is the function that contains the assignment.

This assignment behavior explains the wrong return value of the `sort_priority2` function. The `found` variable is assigned to `True` in the `helper` closure. The closure's assignment is treated as a new variable definition within `helper`, not as an assignment within `sort_priority2`.

```
def sort_priority2(numbers, group):
    found = False         # Scope: 'sort_priority2'
    def helper(x):
        if x in group:
            found = True  # Scope: 'helper' -- Bad!
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

Encountering this problem is sometimes called the *scoping bug* because it can be so surprising to newbies. But this is the intended result. This behavior prevents local variables in a function from polluting the containing module. Otherwise, every assignment within a function would put garbage into the global module scope. Not only would that be noise, but the interplay of the resulting global variables could cause obscure bugs.

Getting Data Out

In Python 3, there is special syntax for getting data out of a closure. The `nonlocal` statement is used to indicate that scope traversal should happen upon assignment for a specific variable name. The only limit is that `nonlocal` won't traverse up to the module-level scope (to avoid polluting globals).

Here, I define the same function again using `nonlocal`:

```
def sort_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

The `nonlocal` statement makes it clear when data is being assigned out of a closure into another scope. It's complementary to the `global` statement, which indicates that a variable's assignment should go directly into the module scope.

However, much like the anti-pattern of global variables, I'd caution against using `nonlocal` for anything beyond simple functions. The side effects of `nonlocal` can be hard to follow. It's especially hard to understand in long functions where the `nonlocal` statements and assignments to associated variables are far apart.

When your usage of `nonlocal` starts getting complicated, it's better to wrap your state in a helper class. Here, I define a class that achieves the same result as the `nonlocal` approach. It's a little longer, but is much easier to read (see Item 23: "Accept Functions for Simple Interfaces Instead of Classes" for details on the __call__ special method).

```
class Sorter(object):
    def __init__(self, group):
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
            return (0, x)
        return (1, x)

sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter.found is True
```

Scope in Python 2

Unfortunately, Python 2 doesn't support the nonlocal keyword. In order to get similar behavior, you need to use a work-around that takes advantage of Python's scoping rules. This approach isn't pretty, but it's the common Python idiom.

```
# Python 2
def sort_priority(numbers, group):
    found = [False]
    def helper(x):
        if x in group:
            found[0] = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found[0]
```

As explained above, Python will traverse up the scope where the found variable is referenced to resolve its current value. The trick is that the value for found is a list, which is mutable. This means that once retrieved, the closure can modify the state of found to send data out of the inner scope (with found[0] = True).

This approach also works when the variable used to traverse the scope is a dictionary, a set, or an instance of a class you've defined.

Things to Remember

✦ Closure functions can refer to variables from any of the scopes in which they were defined.

✦ By default, closures can't affect enclosing scopes by assigning variables.

✦ In Python 3, use the `nonlocal` statement to indicate when a closure can modify a variable in its enclosing scopes.

✦ In Python 2, use a mutable value (like a single-item list) to work around the lack of the `nonlocal` statement.

✦ Avoid using `nonlocal` statements for anything beyond simple functions.

Item 16: Consider Generators Instead of Returning Lists

The simplest choice for functions that produce a sequence of results is to return a list of items. For example, say you want to find the index of every word in a string. Here, I accumulate results in a list using the `append` method and return it at the end of the function:

```
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index + 1)
    return result
```

This works as expected for some sample input.

```
address = 'Four score and seven years ago...'
result = index_words(address)
print(result[:3])

>>>
[0, 5, 11]
```

There are two problems with the `index_words` function.

The first problem is that the code is a bit dense and noisy. Each time a new result is found, I call the `append` method. The method call's bulk (`result.append`) deemphasizes the value being added to the list (`index + 1`). There is one line for creating the result list and another for returning it. While the function body contains ~130 characters (without whitespace), only ~75 characters are important.

A better way to write this function is using a *generator*. Generators are functions that use `yield` expressions. When called, generator functions do not actually run but instead immediately return an iterator. With each call to the `next` built-in function, the iterator will advance the generator to its next `yield` expression. Each value passed to `yield` by the generator will be returned by the iterator to the caller.

Here, I define a generator function that produces the same results as before:

```
def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
```

It's significantly easier to read because all interactions with the result list have been eliminated. Results are passed to `yield` expressions instead. The iterator returned by the generator call can easily be converted to a list by passing it to the `list` built-in function (see Item 9: "Consider Generator Expressions for Large Comprehensions" for how this works).

```
result = list(index_words_iter(address))
```

The second problem with `index_words` is that it requires all results to be stored in the list before being returned. For huge inputs, this can cause your program to run out of memory and crash. In contrast, a generator version of this function can easily be adapted to take inputs of arbitrary length.

Here, I define a generator that streams input from a file one line at a time and yields outputs one word at a time. The working memory for this function is bounded to the maximum length of one line of input.

```
def index_file(handle):
    offset = 0
    for line in handle:
        if line:
            yield offset
        for letter in line:
            offset += 1
            if letter == ' ':
                yield offset
```

Running the generator produces the same results.

```
with open('/tmp/address.txt', 'r') as f:
    it = index_file(f)
    results = islice(it, 0, 3)
    print(list(results))
```

```
>>>
[0, 5, 11]
```

The only gotcha of defining generators like this is that the callers must be aware that the iterators returned are stateful and can't be reused (see Item 17: "Be Defensive When Iterating Over Arguments").

Things to Remember

- ✦ Using generators can be clearer than the alternative of returning lists of accumulated results.
- ✦ The iterator returned by a generator produces the set of values passed to `yield` expressions within the generator function's body.
- ✦ Generators can produce a sequence of outputs for arbitrarily large inputs because their working memory doesn't include all inputs and outputs.

Item 17: Be Defensive When Iterating Over Arguments

When a function takes a list of objects as a parameter, it's often important to iterate over that list multiple times. For example, say you want to analyze tourism numbers for the U.S. state of Texas. Imagine the data set is the number of visitors to each city (in millions per year). You'd like to figure out what percentage of overall tourism each city receives.

To do this you need a normalization function. It sums the inputs to determine the total number of tourists per year. Then it divides each city's individual visitor count by the total to find that city's contribution to the whole.

```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

This function works when given a list of visits.

```
visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)
```

```
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

To scale this up, I need to read the data from a file that contains every city in all of Texas. I define a generator to do this because then I can reuse the same function later when I want to compute tourism numbers for the whole world, a much larger data set (see Item 16: "Consider Generators Instead of Returning Lists").

```
def read_visits(data_path):
    with open(data_path) as f:
        for line in f:
            yield int(line)
```

Surprisingly, calling `normalize` on the generator's return value produces no results.

```
it = read_visits('/tmp/my_numbers.txt')
percentages = normalize(it)
print(percentages)
```

```
>>>
[]
```

The cause of this behavior is that an iterator only produces its results a single time. If you iterate over an iterator or generator that has already raised a `StopIteration` exception, you won't get any results the second time around.

```
it = read_visits('/tmp/my_numbers.txt')
print(list(it))
print(list(it))  # Already exhausted
```

```
>>>
[15, 35, 80]
[]
```

What's confusing is that you also won't get any errors when you iterate over an already exhausted iterator. `for` loops, the `list` constructor, and many other functions throughout the Python standard library expect the `StopIteration` exception to be raised during normal operation. These functions can't tell the difference between an iterator that has no output and an iterator that had output and is now exhausted.

To solve this problem, you can explicitly exhaust an input iterator and keep a copy of its entire contents in a list. You can then iterate over the list version of the data as many times as you need to. Here's the same function as before, but it defensively copies the input iterator:

```
def normalize_copy(numbers):
    numbers = list(numbers)  # Copy the iterator
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Now the function works correctly on a generator's return value.

```
it = read_visits('/tmp/my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)
```

```
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

The problem with this approach is the copy of the input iterator's contents could be large. Copying the iterator could cause your program to run out of memory and crash. One way around this is to accept a function that returns a new iterator each time it's called.

```
def normalize_func(get_iter):
    total = sum(get_iter())   # New iterator
    result = []
    for value in get_iter():  # New iterator
        percent = 100 * value / total
        result.append(percent)
    return result
```

To use `normalize_func`, you can pass in a `lambda` expression that calls the generator and produces a new iterator each time.

```
percentages = normalize_func(lambda: read_visits(path))
```

Though it works, having to pass a lambda function like this is clumsy. The better way to achieve the same result is to provide a new container class that implements the *iterator protocol*.

The iterator protocol is how Python `for` loops and related expressions traverse the contents of a container type. When Python sees a statement like for x in foo it will actually call iter(foo). The iter built-in function calls the foo.__iter__ special method in turn. The __iter__ method must return an iterator object (which itself implements the __next__ special method). Then the for loop repeatedly calls the next built-in function on the iterator object until it's exhausted (and raises a StopIteration exception).

It sounds complicated, but practically speaking you can achieve all of this behavior for your classes by implementing the __iter__ method as a generator. Here, I define an iterable container class that reads the files containing tourism data:

```
class ReadVisits(object):
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield int(line)
```

This new container type works correctly when passed to the original function without any modifications.

```
visits = ReadVisits(path)
percentages = normalize(visits)
print(percentages)

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

This works because the sum method in normalize will call ReadVisits.__iter__ to allocate a new iterator object. The for loop to normalize the numbers will also call __iter__ to allocate a second iterator object. Each of those iterators will be advanced and exhausted independently, ensuring that each unique iteration sees all of the input data values. The only downside of this approach is that it reads the input data multiple times.

Now that you know how containers like ReadVisits work, you can write your functions to ensure that parameters aren't just iterators. The protocol states that when an iterator is passed to the iter built-in function, iter will return the iterator itself. In contrast, when a container type is passed to iter, a new iterator object will be returned each time. Thus, you can test an input value for this behavior and raise a TypeError to reject iterators.

```
def normalize_defensive(numbers):
    if iter(numbers) is iter(numbers):  # An iterator -- bad!
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

This is ideal if you don't want to copy the full input iterator like `normalize_copy` above, but you also need to iterate over the input data multiple times. This function works as expected for `list` and `ReadVisits` inputs because they are containers. It will work for any type of container that follows the iterator protocol.

```
visits = [15, 35, 80]
normalize_defensive(visits)  # No error
visits = ReadVisits(path)
normalize_defensive(visits)  # No error
```

The function will raise an exception if the input is iterable but not a container.

```
it = iter(visits)
normalize_defensive(it)
```

```
>>>
TypeError: Must supply a container
```

Things to Remember

- ✦ Beware of functions that iterate over input arguments multiple times. If these arguments are iterators, you may see strange behavior and missing values.
- ✦ Python's iterator protocol defines how containers and iterators interact with the `iter` and `next` built-in functions, `for` loops, and related expressions.
- ✦ You can easily define your own iterable container type by implementing the `__iter__` method as a generator.
- ✦ You can detect that a value is an iterator (instead of a container) if calling `iter` on it twice produces the same result, which can then be progressed with the `next` built-in function.

Item 18: Reduce Visual Noise with Variable Positional Arguments

Accepting optional positional arguments (often called *star args* in reference to the conventional name for the parameter, `*args`) can make a function call more clear and remove *visual noise*.

For example, say you want to log some debug information. With a fixed number of arguments, you would need a function that takes a message and a list of values.

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('My numbers are', [1, 2])
log('Hi there', [])

>>>
My numbers are: 1, 2
Hi there
```

Having to pass an empty list when you have no values to log is cumbersome and noisy. It'd be better to leave out the second argument entirely. You can do this in Python by prefixing the last positional parameter name with *. The first parameter for the log message is required, whereas any number of subsequent positional arguments are optional. The function body doesn't need to change, only the callers do.

```
def log(message, *values):  # The only difference
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))

log('My numbers are', 1, 2)
log('Hi there')  # Much better

>>>
My numbers are: 1, 2
Hi there
```

If you already have a list and want to call a variable argument function like `log`, you can do this by using the `*` operator. This instructs Python to pass items from the sequence as positional arguments.

```
favorites = [7, 33, 99]
log('Favorite colors', *favorites)

>>>
Favorite colors: 7, 33, 99
```

There are two problems with accepting a variable number of positional arguments.

The first issue is that the variable arguments are always turned into a tuple before they are passed to your function. This means that if the caller of your function uses the `*` operator on a generator, it will be iterated until it's exhausted. The resulting tuple will include every value from the generator, which could consume a lot of memory and cause your program to crash.

```
def my_generator():
    for i in range(10):
        yield i

def my_func(*args):
    print(args)

it = my_generator()
my_func(*it)

>>>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Functions that accept `*args` are best for situations where you know the number of inputs in the argument list will be reasonably small. It's ideal for function calls that pass many literals or variable names together. It's primarily for the convenience of the programmer and the readability of the code.

The second issue with `*args` is that you can't add new positional arguments to your function in the future without migrating every caller. If you try to add a positional argument in the front of the argument list, existing callers will subtly break if they aren't updated.

```
def log(sequence, message, *values):
    if not values:
        print('%s: %s' % (sequence, message))
    else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s: %s' % (sequence, message, values_str))
```

```
log(1, 'Favorites', 7, 33)      # New usage is OK
log('Favorite numbers', 7, 33)  # Old usage breaks
```

```
>>>
1: Favorites: 7, 33
Favorite numbers: 7: 33
```

The problem here is that the second call to `log` used `7` as the `message` parameter because a `sequence` argument wasn't given. Bugs like this are hard to track down because the code still runs without raising any exceptions. To avoid this possibility entirely, you should use keyword-only arguments when you want to extend functions that accept `*args` (see Item 21: "Enforce Clarity with Keyword-Only Arguments").

Things to Remember

- ✦ Functions can accept a variable number of positional arguments by using `*args` in the `def` statement.
- ✦ You can use the items from a sequence as the positional arguments for a function with the `*` operator.
- ✦ Using the `*` operator with a generator may cause your program to run out of memory and crash.
- ✦ Adding new positional parameters to functions that accept `*args` can introduce hard-to-find bugs.

Item 19: Provide Optional Behavior with Keyword Arguments

Like most other programming languages, calling a function in Python allows for passing arguments by position.

```
def remainder(number, divisor):
    return number % divisor

assert remainder(20, 7) == 6
```

All positional arguments to Python functions can also be passed by keyword, where the name of the argument is used in an assignment within the parentheses of a function call. The keyword arguments can be passed in any order as long as all of the required positional arguments are specified. You can mix and match keyword and positional arguments. These calls are equivalent:

```
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, number=20)
```

Positional arguments must be specified before keyword arguments.

```
remainder(number=20, 7)

>>>
SyntaxError: non-keyword arg after keyword arg
```

Each argument can only be specified once.

```
remainder(20, number=7)

>>>
TypeError: remainder() got multiple values for argument
➥'number'
```

The flexibility of keyword arguments provides three significant benefits.

The first advantage is that keyword arguments make the function call clearer to new readers of the code. With the call `remainder(20, 7)`, it's not evident which argument is the number and which is the divisor without looking at the implementation of the `remainder` method. In the call with keyword arguments, `number=20` and `divisor=7` make it immediately obvious which parameter is being used for each purpose.

The second impact of keyword arguments is that they can have default values specified in the function definition. This allows a function to provide additional capabilities when you need them but lets you accept the default behavior most of the time. This can eliminate repetitive code and reduce noise.

For example, say you want to compute the rate of fluid flowing into a vat. If the vat is also on a scale, then you could use the difference between two weight measurements at two different times to determine the flow rate.

```
def flow_rate(weight_diff, time_diff):
    return weight_diff / time_diff

weight_diff = 0.5
time_diff = 3
flow = flow_rate(weight_diff, time_diff)
print('%.3f kg per second' % flow)

>>>
0.167 kg per second
```

In the typical case, it's useful to know the flow rate in kilograms per second. Other times, it'd be helpful to use the last sensor measurements to approximate larger time scales, like hours or days. You can provide this behavior in the same function by adding an argument for the time period scaling factor.

```python
def flow_rate(weight_diff, time_diff, period):
    return (weight_diff / time_diff) * period
```

The problem is that now you need to specify the `period` argument every time you call the function, even in the common case of flow rate per second (where the period is 1).

```python
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```

To make this less noisy, I can give the `period` argument a default value.

```python
def flow_rate(weight_diff, time_diff, period=1):
    return (weight_diff / time_diff) * period
```

The `period` argument is now optional.

```python
flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

This works well for simple default values (it gets tricky for complex default values—see Item 20: "Use `None` and Docstrings to Specify Dynamic Default Arguments").

The third reason to use keyword arguments is that they provide a powerful way to extend a function's parameters while remaining backwards compatible with existing callers. This lets you provide additional functionality without having to migrate a lot of code, reducing the chance of introducing bugs.

For example, say you want to extend the `flow_rate` function above to calculate flow rates in weight units besides kilograms. You can do this by adding a new optional parameter that provides a conversion rate to your preferred measurement units.

```python
def flow_rate(weight_diff, time_diff,
              period=1, units_per_kg=1):
    return ((weight_diff / units_per_kg) / time_diff) * period
```

The default argument value for `units_per_kg` is 1, which makes the returned weight units remain as kilograms. This means that all existing callers will see no change in behavior. New callers to `flow_rate` can specify the new keyword argument to see the new behavior.

```python
pounds_per_hour = flow_rate(weight_diff, time_diff,
                            period=3600, units_per_kg=2.2)
```

The only problem with this approach is that optional keyword arguments like `period` and `units_per_kg` may still be specified as positional arguments.

```
pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)
```

Supplying optional arguments positionally can be confusing because it isn't clear what the values 3600 and 2.2 correspond to. The best practice is to always specify optional arguments using the keyword names and never pass them as positional arguments.

Note

Backwards compatibility using optional keyword arguments like this is crucial for functions that accept *args (see Item 18: "Reduce Visual Noise with Variable Positional Arguments"). But an even better practice is to use keyword-only arguments (see Item 21: "Enforce Clarity with Keyword-Only Arguments").

Things to Remember

- ✦ Function arguments can be specified by position or by keyword.
- ✦ Keywords make it clear what the purpose of each argument is when it would be confusing with only positional arguments.
- ✦ Keyword arguments with default values make it easy to add new behaviors to a function, especially when the function has existing callers.
- ✦ Optional keyword arguments should always be passed by keyword instead of by position.

Item 20: Use None and Docstrings to Specify Dynamic Default Arguments

Sometimes you need to use a non-static type as a keyword argument's default value. For example, say you want to print logging messages that are marked with the time of the logged event. In the default case, you want the message to include the time when the function was called. You might try the following approach, assuming the default arguments are reevaluated each time the function is called.

```
def log(message, when=datetime.now()):
    print('%s: %s' % (when, message))

log('Hi there!')
sleep(0.1)
log('Hi again!')
```

```
>>>
2014-11-15 21:10:10.371432: Hi there!
2014-11-15 21:10:10.371432: Hi again!
```

The timestamps are the same because datetime.now is only executed a single time: when the function is defined. Default argument values are evaluated only once per module load, which usually happens when a program starts up. After the module containing this code is loaded, the datetime.now default argument will never be evaluated again.

The convention for achieving the desired result in Python is to provide a default value of None and to document the actual behavior in the docstring (see Item 49: "Write Docstrings for Every Function, Class, and Module"). When your code sees an argument value of None, you allocate the default value accordingly.

```python
def log(message, when=None):
    """Log a message with a timestamp.

    Args:
        message: Message to print.
        when: datetime of when the message occurred.
            Defaults to the present time.
    """
    when = datetime.now() if when is None else when
    print('%s: %s' % (when, message))
```

Now the timestamps will be different.

```python
log('Hi there!')
sleep(0.1)
log('Hi again!')

>>>
2014-11-15 21:10:10.472303: Hi there!
2014-11-15 21:10:10.573395: Hi again!
```

Using None for default argument values is especially important when the arguments are mutable. For example, say you want to load a value encoded as JSON data. If decoding the data fails, you want an empty dictionary to be returned by default. You might try this approach.

```python
def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default
```

The problem here is the same as the datetime.now example above. The dictionary specified for default will be shared by all calls to decode because default argument values are only evaluated once (at module load time). This can cause extremely surprising behavior.

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)

>>>
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

You'd expect two different dictionaries, each with a single key and value. But modifying one seems to also modify the other. The culprit is that `foo` and `bar` are both equal to the `default` parameter. They are the same dictionary object.

```
assert foo is bar
```

The fix is to set the keyword argument default value to `None` and then document the behavior in the function's docstring.

```
def decode(data, default=None):
    """Load JSON data from a string.

    Args:
        data: JSON data to decode.
        default: Value to return if decoding fails.
            Defaults to an empty dictionary.
    """
    if default is None:
        default = {}
    try:
        return json.loads(data)
    except ValueError:
        return default
```

Now, running the same test code as before produces the expected result.

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)

>>>
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

Things to Remember

- ✦ Default arguments are only evaluated once: during function definition at module load time. This can cause odd behaviors for dynamic values (like {} or []).
- ✦ Use None as the default value for keyword arguments that have a dynamic value. Document the actual default behavior in the function's docstring.

Item 21: Enforce Clarity with Keyword-Only Arguments

Passing arguments by keyword is a powerful feature of Python functions (see Item 19: "Provide Optional Behavior with Keyword Arguments"). The flexibility of keyword arguments enables you to write code that will be clear for your use cases.

For example, say you want to divide one number by another but be very careful about special cases. Sometimes you want to ignore ZeroDivisionError exceptions and return infinity instead. Other times, you want to ignore OverflowError exceptions and return zero instead.

```python
def safe_division(number, divisor, ignore_overflow,
                  ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

Using this function is straightforward. This call will ignore the float overflow from division and will return zero.

```python
result = safe_division(1, 10**500, True, False)
print(result)
```

```
>>>
0.0
```

This call will ignore the error from dividing by zero and will return infinity.

```
result = safe_division(1, 0, False, True)
print(result)
```

```
>>>
inf
```

The problem is that it's easy to confuse the position of the two Boolean arguments that control the exception-ignoring behavior. This can easily cause bugs that are hard to track down. One way to improve the readability of this code is to use keyword arguments. By default, the function can be overly cautious and can always re-raise exceptions.

```
def safe_division_b(number, divisor,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    # ...
```

Then callers can use keyword arguments to specify which of the ignore flags they want to flip for specific operations, overriding the default behavior.

```
safe_division_b(1, 10**500, ignore_overflow=True)
safe_division_b(1, 0, ignore_zero_division=True)
```

The problem is, since these keyword arguments are optional behavior, there's nothing forcing callers of your functions to use keyword arguments for clarity. Even with the new definition of `safe_division_b`, you can still call it the old way with positional arguments.

```
safe_division_b(1, 10**500, True, False)
```

With complex functions like this, it's better to require that callers are clear about their intentions. In Python 3, you can demand clarity by defining your functions with keyword-only arguments. These arguments can only be supplied by keyword, never by position.

Here, I redefine the `safe_division` function to accept keyword-only arguments. The * symbol in the argument list indicates the end of positional arguments and the beginning of keyword-only arguments.

```
def safe_division_c(number, divisor, *,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    # ...
```

Now, calling the function with positional arguments for the keyword arguments won't work.

```
safe_division_c(1, 10**500, True, False)
```

```
>>>
TypeError: safe_division_c() takes 2 positional arguments but
➥4 were given
```

Keyword arguments and their default values work as expected.

```
safe_division_c(1, 0, ignore_zero_division=True)  # OK

try:
    safe_division_c(1, 0)
except ZeroDivisionError:
    pass  # Expected
```

Keyword-Only Arguments in Python 2

Unfortunately, Python 2 doesn't have explicit syntax for specifying keyword-only arguments like Python 3. But you can achieve the same behavior of raising TypeErrors for invalid function calls by using the ** operator in argument lists. The ** operator is similar to the * operator (see Item 18: "Reduce Visual Noise with Variable Positional Arguments"), except that instead of accepting a variable number of positional arguments, it accepts any number of keyword arguments, even when they're not defined.

```
# Python 2
def print_args(*args, **kwargs):
    print 'Positional:', args
    print 'Keyword:   ', kwargs

print_args(1, 2, foo='bar', stuff='meep')
```

```
>>>
Positional: (1, 2)
Keyword:    {'foo': 'bar', 'stuff': 'meep'}
```

To make safe_division take keyword-only arguments in Python 2, you have the function accept **kwargs. Then you pop keyword arguments that you expect out of the kwargs dictionary, using the pop method's second argument to specify the default value when the key is missing. Finally, you make sure there are no more keyword arguments left in kwargs to prevent callers from supplying arguments that are invalid.

```
# Python 2
def safe_division_d(number, divisor, **kwargs):
    ignore_overflow = kwargs.pop('ignore_overflow', False)
    ignore_zero_div = kwargs.pop('ignore_zero_division', False)
    if kwargs:
        raise TypeError('Unexpected **kwargs: %r' % kwargs)
    # ...
```

Now, you can call the function with or without keyword arguments.

```
safe_division_d(1, 10)
safe_division_d(1, 0, ignore_zero_division=True)
safe_division_d(1, 10**500, ignore_overflow=True)
```

Trying to pass keyword-only arguments by position won't work, just like in Python 3.

```
safe_division_d(1, 0, False, True)
```

```
>>>
TypeError: safe_division_d() takes 2 positional arguments but 4
➥were given
```

Trying to pass unexpected keyword arguments also won't work.

```
safe_division_d(0, 0, unexpected=True)
```

```
>>>
TypeError: Unexpected **kwargs: {'unexpected': True}
```

Things to Remember

- Keyword arguments make the intention of a function call more clear.
- Use keyword-only arguments to force callers to supply keyword arguments for potentially confusing functions, especially those that accept multiple Boolean flags.
- Python 3 supports explicit syntax for keyword-only arguments in functions.
- Python 2 can emulate keyword-only arguments for functions by using `**kwargs` and manually raising `TypeError` exceptions.

Index

Symbols

%r, for printable strings, 203
%s, for human-readable strings, 202
* operator, liability of, 44–45
* symbol, for keyword-only arguments, 52–53
*args
  optional keyword arguments and, 48
  variable positional arguments and, 43–45
**kwargs, for keyword-only arguments, 53–54

A

__all__ special attribute
  avoiding, 183
  listing all public names, 181–183
ALL_CAPS format, 3
Allocation of memory, tracemalloc module and, 214–216
APIs (application programming interfaces)
  future-proofing, 186–187
  internal, allowing subclass access to, 80–82
  packages providing stable, 181–184
  root exceptions and, 184–186
  using functions for, 61–64
append method, 36–37
Arguments
  defensively iterating over, 38–42
  keyword, 45–48
  keyword-only, 51–54
  optional positional, 43–45
as clauses, in renaming modules, 181
as targets, with statements and, 155–156
assertEqual helper method, verifying equality, 206
assertRaises helper method, verifying exceptions, 206
assertTrue helper method, for Boolean expressions, 206
asyncio built-in module, vs. blocking I/O, 125
AttributeError exception raising, 102–103
Attribute(s). *See also* Private attributes; Public attributes
  adding missing default values, 159–160
  lazily loading/saving, 100–105
  metaclasses annotating, 112–115
  names, conflicts over, 81–82

B

Binary mode, for reading/writing data, 7
Binary tree class, inheriting from collections.abc, 84–86
bisect module, for binary searches, 169
Blocking operations, in Queue class, 132–136
Bookkeeping
  with dictionaries, 55–58
  helper classes for, 58–60
bt command, of interactive debugger, 208
Buffer sizes, in Queue class, 132–136
Bytecode, interpreter state for, 122
bytes instances, for character sequences, 5–7

C

__call__ special method, with instances, 63–64
callable built-in function, 63
CapitalizedWord format, 3

Central processing unit. *See* CPU (central processing unit)
C-extension modules
 for CPU bottlenecks, 145
 problems with, 146
chain function, of itertools module, 170
Child classes, initializing parent classes from, 69–73
Child processes, subprocess managing, 118–121
Circular dependencies
 dynamic imports resolving, 191–192
 import reordering for, 189–190
 import/configure/run steps for, 190–191
 in importing modules, 187–188
 refactoring code for, 189
Clarity, with keyword arguments, 51–54
Class interfaces
 @property method improving, 91–94
 use public attributes for defining, 87–88
class statements, metaclasses receiving, 106–107
__class__ variable
 registering classes and, 108–112
 super built_in function with, 73
Classes. *See also* Metaclasses; Subclasses
 annotating properties of, 112–115
 for bookkeeping, 58–60
 docstrings for, 177
 initializing parent, 69–73
 metaclasses registering, 108–112
 mix-in, 73–78
 versioning, 160–161
@classmethod
 in accessing private attributes, 78–79
 polymorphism, for constructing objects generically, 67–69
Closures, interacting with variable scope, 31–36
collections module
 defaultdict class from, 168
 deque class from, 166–167
 OrderedDict class from, 167–168
collections.abc module, custom containers inheriting from, 84–86
combination function, of itertools module, 170
Command-lines
 correct Python version, 1, 2
 starting child processes, 119–120
communicate method
 reading child process output, 118–119
 timeout parameter with, 121
Community-built modules, Python Package Index for, 173–174
Complex expressions, helper functions and, 8–10
Concurrency
 coroutines and, 137–138
 defined, 117
 in pipelines, 129–132
 Queue class and, 132–136
concurrent.futures built-in module, enabling parallelism, 146–148
configparser built-in module, for production configuration, 201
Containers
 inheriting from collections.abc, 84–86
 iterable, 41–42
contextlib built-in module, enabling with statements, 154–155
contextmanager decorator
 purpose of, 154
 as targets and, 155–156
continue command, of interactive debugger, 209
Conway's Game of Life, coroutines and, 138–143
Coordinated Universal Time (UTC), in time conversions, 162–165
copyreg built-in module
 adding missing attribute values, 159–160
 controlling pickle behavior, 158
 providing stable import paths, 161–162
 versioning classes with, 160–161
Coroutines
 in Conway's Game of Life, 138–143
 purpose of, 137–138
 in Python 2, 143–145
count method, for custom container types, 85–86
cProfile module, for accurate profiling, 210–213
CPU (central processing unit)
 bottleneck difficulties, 145–146
 time, threads wasting, 131–132
 usage, child processes and, 118–121
CPython interpreter, effect of GIL on, 122–123

CPython runtime
memory management with, 214
cumtime column, in profiler statistics, 211
cumtime percall column, in profiler statistics, 211
cycle function, of itertools module, 170

D

Data models, @property improving, 91–95
Data races, Lock preventing, 126–129
datetime built-in module, for time conversions, 164–166
deactivate command, disabling pyvenv tool, 195–196
Deadlocks, timeout parameter avoiding, 121
Deallocation of memory, tracemalloc managing, 214–216
Debuggers, decorator problems with, 151, 153
Debugging
interactive, with pdb module, 208–209
memory usage, 214–216
print function and, 202
repr strings for, 202–204
root exceptions for, 185–186
Decimal class, for numerical precision, 171–173
Decorators, functionality of, 151–153
functools, 151–153
Default arguments
approach to serialization, 159–160
namedtuple classes and, 59
using dynamic values for, 48–51
Default value hooks, 62–64
defaultdict using, 62–64
Default values
copyreg built-in module and, 159–160
of keyword arguments, 46–47
defaultdict class, for dictionaries, 168
Dependencies
circular, 187–192
reproducing, 196–197
transitive, 192–194
Dependency injection, 191
Deployment environments, module-scoped code for, 199–201
deque class, as double-ended queue, 166–167
Descriptors
enabling reusable property logic, 90
in modifying class properties, 112–115
for reusable @property methods, 97–100
Deserializing objects
default attribute values and, 159–160
pickle built-in module for, 157–158
stable import paths and, 161–162
Development environment, unique configurations/assumptions for, 199–201
Diamond inheritance, initializing parent classes and, 70–71
__dict__ attribute, viewing object internals, 204
Dictionaries
bookkeeping with, 55–58
comprehension expressions in, 16
default, 168
ordered, 167–168
translating related objects into, 74–75
__doc__ special attribute, retrieving docstrings, 175–176
Docstrings
class-level, 177
documenting default behavior in, 48–51
for functions, 178–179
importance/placement of, 175–176
module, 176–177
doctest built-in module, 179
Documentation
docstrings for. *See* Docstrings
importance of, 175
Documentation-generation tools, 176
Double-ended queues, deque classes as, 166–167
__double_leading_underscore format, 3
down command, of interactive debugger, 209
dropwhile function, of itertools module, 170
Dynamic imports
avoiding, 192
resolving circular dependencies, 191–192
Dynamic state, defined, 55

E
else blocks
  after for/while loops, 23–25
  during exception handling, 26–27
end indexes, in slicing sequences, 10–13
__enter__ method, in defining new classes, 154
enumerate built-in function, preferred features of, 20–21
environ dictionary, tailoring modules with, 201
eval built-in function, for re-creating original values, 203
Exceptions
  raising, 29–31
  root, 184–187
  try/finally blocks and, 26–28
Execution time, optimization of, 209–213
__exit__ method, in defining new classes, 154
Expressions
  in list comprehensions, 16–18
  PEP 8 guidance for, 4

F
filter built-in function, list comprehensions vs., 15–16
filterfalse function, of itertools module, 170
finally blocks, during exception handling, 26–27
First-in-first-out queues, deque class for, 166–167
for loops
  else blocks after, 23–25
  iterator protocol and, 40–42
Fraction class, for numerical precision, 172
Functions
  closure/variable scope interaction, 31–36
  decorated, 151–153
  docstrings for, 178–179
  exceptions vs. return None, 29–31
  as first-class objects, 32, 63–64
  generator vs. returning lists, 36–38
  iterating over arguments, 38–42
  keyword arguments for, 45–48
  keyword-only arguments for, 51–54
  optional positional arguments for, 43–45
  for simple interfaces, 61–64
  simultaneous, coroutines for, 137–138
functools built-in module, for defining decorators, 152–153

G
Game of Life, coroutines in, 138–143
Garbage collector, cleanup by, 99
gc built-in module, debugging memory usage, 214–215
Generator(s)
  coroutine extensions of, 137–138
  expressions, for large comprehensions, 18–20
  returning lists vs., 36–38
Generic class method, for constructing objects, 67–69
Generic functionality, with mix-in classes, 74–78
__get__ method, for descriptor protocol, 97–100
__getattr__ special method, to lazily load attributes, 100–103
__getattribute__ method, accessing instance variables in, 104–105
__getattribute__ method, descriptor protocol and, 98–100
__getattribute__ special method, for repeated access, 102–105
__getitem__ special method
  custom implementation of, 84–86
  in slicing sequences, 10
Getter methods
  descriptor protocol for, 98–100
  problems with using, 87–88
  providing with @property, 88–89
GIL (global interpreter lock)
  corruption of data structures and, 126–127
  defined, 122
  preventing parallelism in threads, 122–125, 145, 146–147
Global scope, 33

H
hasattr built-in function, determining existence of properties, 103
hashlib built-in module, 120
heappop function, for priority queues, 168–169
heappush function, for priority queues, 168–169

heapq module, for priority queues, 168–169
help function
  decorator problems with, 152–153
  in interactive debugger, 208
Helper classes
  for bookkeeping, 58–60
  providing stateful closure behavior, 62–63
Helper functions, complex expressions into, 8–10
Hooks
  to access missing attributes, 100–105
  default value, 62–64
  functions acting as, 61–62
  in modifying class properties, 113

I

IEEE 754 (IEEE Standard for Floating-Point Arithmetic), 171–172
if/else expressions, for simplification, 9–10
import * statements
  avoiding, 183–184
  in providing stable APIs, 182–183
Import paths, stable, copyreg providing, 161–162
Import reordering, for circular dependencies, 189–190
import statements
  as dynamic imports, 191–192
  with packages, 180–181
Incrementing in place, public attributes for, 88
index method, for custom container types, 85–86
Infinite recursion, super() function avoiding, 101–105
Inheritance
  from collections.abc, 84–86
  method resolution order (MRO) and, 71
  multiple, for mix-in utility classes, 77–78
__init__ method
  as single constructor per class, 67, 69
  initializing parent class, 69–71
__init__.py
  defining packages, 180
  modifying, 182
Initializing parent classes
  __init__ method for, 69–71
  method resolution order (MRO) and, 71
  super built-in function for, 70–73
Integration tests, 207
Interactive debugging, with pdb, 208–209
Intermediate root exceptions, future-proofing APIs, 186–187
I/O (input/output)
  between child processes, 118–121
  threads for blocking I/O, 124–125
IOError, except blocks and, 26–27
IronPython runtime, 1, 2
isinstance
  bytes/str/unicode and, 5–6
  with coroutines, 142
  dynamic type inspection with, 74–75
  metaclasses and, 114
  pickle module and, 158
  testing and, 205
islice function, of itertools module, 170
iter built-in function, 41–42
__iter__ method
  as generator, 41–42
  iterable container class, defined, 41–42
Iterator protocol, 40–42
Iterators
  as function arguments, 39
  generators returning, 37–38
  zip function processing, 21–23
itertools built-in module
  functions of, 169–170
izip_longest function, for iterating in parallel, 23

J

join method, of Queue class, 132–136
Jython runtime, 1, 2

K

Keyword arguments
  constructing classes with, 58
  dynamic default argument values, 48–51
  providing optional behavior, 45–48
Keyword-only arguments
  for clarity, 51–53
  in Python 2, 53–54

L

lambda expression
  as key hook, 61
  vs. list comprehensions, 15–16
  producing iterators and, 40
  in profiling, 210–212
Language hooks, for missing attributes, 100–105
Lazy attributes, __getattr__/__setattr__/__getattribute__ for, 100–105
_leading_underscore format, 3
Leaky bucket quota, implementing, 92–95
len built-in function, for custom sequence types, 85
__len__ special method, for custom sequence types, 85
list built-in type, performance as FIFO queue , 166–167
List comprehensions
  generator expressions for, 18–20
  instead of map/filter, 15–16
  number of expressions in, 16–18
list type, subclassing, 83–84
Lists, slicing, 10–13
locals built-in function, 152, 208
localtime function, from time module, 163–164
Lock class
  preventing data races, 126–129
  in with statements, 153–154
Logging
  debug function for, 154–156
  severity levels, 154–155
Loops
  else blocks after, 23–25
  in list comprehensions, 16–18
  range/enumerate functions, 20–21
lowercase_underscore format, 3

M

map built-in function, list comprehensions vs., 15–16
Memory
  coroutine use of, 137
  threads requiring, 136
Memory leaks
  by descriptor classes, 99–100
  identifying, 214–216
Memory management, with tracemalloc module, 214–216
Meta.__new__ method
  in metaclasses, 107
  setting class attributes, 114
__metaclass__ attribute, in Python 2, 106–107
Metaclasses
  annotating attributes with, 112–115
  for class registration, 108–112
  defined, 87, 106
  validating subclasses, 105–108
method resolution order (MRO), for superclass initialization order, 70–73
Mix-in classes
  composing from simple behaviors, 74–75
  defined, 73–74
  pluggable behaviors for, 75–76
  utility, creating hierachies of, 77–78
mktime, for time conversion, 163, 165
Mock functions and classes
  unittest.mock built-in module, 206
__module__ attribute, 106, 153
Modules
  breaking circular dependencies in, 187–192
  community-built, 173–174
  docstrings, 176–177
  packages for organizing, 179–184
  providing stable APIs from, 181–184
  tailoring for deployment environment, 199–201
Module-scoped code, for deployment environments, 199–201
MRO (method resolution order), for superclass initialization order, 70–73
Multiple conditions, in list comprehensions, 16–18
Multiple inheritance, for mix-in utility classes, 73–78
Multiple iterators, zip built-in function and, 21–23
Multiple loops, in list comprehensions, 16–18
multiprocessing built-in module, enabling parallelism, 146–148

Mutual-exclusion locks (mutex)
    GIL as, 122
    Lock class as, 126–129
    in with statements, 153–154

N

__name__ attribute in defining decorators, 151, 153
    in registering classes, 109–110
    testing and, 206
namedtuple type
    defining classes, 58
    limitations of, 59
NameError exception, 33
Namespace packages, with Python 3.4, 180
Naming conflicts, private attributes to avoid, 81–82
Naming styles, 3–4
ncalls column in profiler statistics, 211
__new__ method, of metaclasses, 106–108
next built-in function, 41–42
next command, of interactive debugger, 209
__next__ special method, iterator object implementing, 41
Noise reduction, keyword arguments and, 45–48
None value
    functions returning, 29–31
    specifying dynamic default values, 48–51
nonlocal statement, in closures modifying variables, 34–35
nsmallest function, for priority queues, 168–169
Numerical precision, with Decimal class, 171–173

O

Objects, accessing missing attributes in, 100–105
On-the-fly calculations, using @property for, 91–95
Optimization, profiling prior to, 209–213
Optional arguments
    keyword, 47–48
    positional, 43–45
OrderedDict class, for dictionaries, 167–168
OverflowError exceptions, 51

P

Packages
    dividing modules into namespaces, 180–181
    as modules containing modules, 179–180
    providing stable APIs with, 181–184
Parallelism
    avoiding threads for, 122–123
    child processes and, 118–121
    concurrent.futures for true, 146–148
    corruption of data structures and, 126–128
    defined, 117
    need for, 145–146
Parent classes
    accessing private attributes of, 79–81
    initializing, 70–73
pdb built-in module, for interactive debugging, 208–209
pdb.set_trace() statements, 208–209
PEP 8 (Python Enhancement Proposal #8) style guide
    expression/statement rules, 4
    naming styles in, 3–4, 80
    overview of, 2–3
    whitespace rules, 3
permutations function, of itertools module, 170
pickle built-in module
    adding missing attribute values, 159–160
    providing stable import paths for, 161–162
    serializing/deserializing objects, 157–158
    versioning classes for, 160–161
pip command-line tool
    reproducing environments, 196–197
    transitive dependencies and, 192–193
    for utilizing Package Index, 173
pip freeze command, saving package dependencies, 196
Pipelines
    concurrency in, 129–131
    problems with, 132
    Queue class building, 132–136
Polymorphism
    @classmethods utilizing, 65–69
    defined, 64

Popen constructor, starting child processes, 118
Positional arguments
constructing classes with, 58
keyword arguments and, 45–48
reducing visual noise, 43–45
print function, for debugging output, 202–203, 208
print_stats output, for profiling, 213
Printable representation, repr function for, 202–204
Private attributes
accessing, 78–80
allowing subclass access to, 81–83
indicating internal APIs, 80
ProcessPoolExecutor class, enabling parallelism, 147–148
product function, of itertools module, 170
Production environment, unique configurations for, 199–201
profile module, liabilities of, 210
@property method
defining special behavior with, 88–89
descriptors for reusing, 97–100
giving attributes new functionality, 91–94
improving data models with, 95
numerical attributes, into on-the-fly calculations, 91–95
problems with overusing, 95–96
unexpected side effects in, 90–91
@property.setter, modifying object state in, 91
pstats built-in module, extracting statistics, 211
Public attributes
accessing, 78
defining new class interfaces with, 87–88
giving new functionality to, 91–94
preferred features of, 80–82
Pylint tool, for Python source code, 4
PyPI (Python Package Index), for community-built modules, 173–174
PyPy runtime, 1, 2
Python 2
coroutines in, 143–145
determining use of, 2
keyword-only arguments in, 53–54
metaclass syntax in, 106–107
mutating closure variables in, 35
str and unicode in, 5–7
zip built-in function in, 22
Python 3
class decorators in, 111
determining use of, 2
closures and nonlocal statements in, 34–35
keyword-only arguments in, 51–53
metaclass syntax in, 106
str and bytes in, 5–7
Python Enhancement Proposal #8. *See* PEP 8 (Python Enhancement Proposal #8) style guide
Python Package Index (PyPI), for community-built modules, 173–174
Python threads. *See* Threads
pytz module
installing, 173
pyvenv tool and, 194
for time conversions, 165–166
pyvenv command-line tool
purpose of, 194
reproducing environments, 196–197
for virtual environments, 194–196

Q

quantize method, of Decimal class, for numerical data, 172
Queue class, coordinating work between threads, 132–136

R

range built-in function, in loops, 20
Read the Docs community-funded site, 176
Refactoring attributes, @property instead of, 91–95
Refactoring code, for circular dependencies, 189
Registering classes, metaclasses for, 108–112
Repetitive code
composing mix-ins to minimize, 74
keyword arguments eliminating, 45–48
__repr__ special method, customizing class printable representation, 203–204
repr strings, for debugging output, 202–204
requirements.txt file, for installing packages, 197
return command, of interactive debugger, 209

return statements
  in generators, 140
  not allowed in Python 2 generators, 144
Root exceptions
  finding bugs in code with, 185–186
  future-proofing APIs, 186–187
  insulating callers from APIs, 184–185
Rule of least surprise, 87, 90, 91
runcall method, for profiling, 211–213

S

Scopes, variable, closure interaction with, 31–36
Scoping bug, in closures, 34
select built-in module, blocking I/O, 121, 124
Serializing, data structures, 109
Serializing objects, pickle and
  default argument approach to, 159–160
  default attribute values and, 159–160
  pickle built-in module for, 157–158
  stable import paths and, 161–162
__set__ method, for descriptor protocol, 97–100
set_trace function, pdb module running, 208–209
setattr built-in function
  annotating class attributes and, 113
  in bad thread interactions, 127–128
  lazy attributes and, 101–102, 104
__setattr__ special method, to lazily set attributes, 103–105
__setitem__ special method, in slicing sequences, 10
Sets, comprehension expressions in, 16
setter attribute, for @property method, 88–89
Setter methods
  descriptor protocol for, 98–100
  liability of using, 87–88
  providing with @property, 88–89
setuptools, in virtual environments, 195–197
Single constructor per class, 67, 69
Single-line expressions, difficulties with, 8–10
six tool, in adopting Python 3, 2
Slicing sequences
  basic functions of, 10–13
  stride syntax in, 13–15
Sort, key argument, closure functions as, 31–32
source bin/activate command, enabling pyvenv tool, 195
Speedup, concurrency vs. parallelism for, 117
Star args (*args), 43
start indexes, in slicing sequences, 10–13
Statements, PEP 8 guidance for, 4
Static type checking, lack of, 204–205
Stats object, for profiling information, 211–213
step command, of interactive debugger, 209
StopIteration exception, 39, 41
str instances, for character sequences, 5–7
stride syntax, in slicing sequences, 13–15
strptime functions, conversion to/from local time, 163–164
Subclasses
  allowing access to private fields, 81–83
  constructing/connecting generically, 65–69
  list type, 83–84
  TestCase, 206–207
  validating with metaclasses, 105–108
subprocess built-in module, for child processes, 118–121
super built-in function, initializing parent classes, 71–73
super method, avoiding infinite recursion, 101–105
Superclass initialization order, MRO resolving, 70–73
Syntax
  decorators, 151–153
  for closures mutating variables, 34–35
  for keyword-only arguments, 52–53
  loops with else blocks, 23
  list comprehensions, 15
  metaclasses, 106
  slicing, 10–13
SyntaxError exceptions, dynamic imports and, 192
sys module, guiding module definitions, 201
System calls, blocking I/O and, 124–125

T

takewhile function, of itertools module, 170
task_done call, method of the Queue class, in building pipelines, 134
tee function, of itertools module, 170
Test methods, 206–207
TestCase classes, subclassing, 206–207
threading built-in module, Lock class in, 126–129
ThreadPoolExecutor class, not enabling parallelism, 147–148
Threads
- blocking I/O and, 124–125
- coordinating work between, 132–136
- parallelism prevented by, 122–123, 145, 146–147
- preventing data races between, 126–129
- problems with, 136
- usefulness of multiple, 124

time built-in module, limitations of, 163–164
Time zone conversion methods, 162–166
timeout parameter, in child process I/O, 121
tottime column, in profiler statistics, 211
tottime percall column, in profiler statistics, 211
tracemalloc built-in module, for memory optimization, 214–216
Transitive dependencies, 192–194
try/except statements, root exceptions and, 185
try/except/else/finally blocks, during exception handling, 27–28
try/finally blocks
- during exception handling, 26–27
- with statements providing reusable, 154–155

Tuples
- extending, 58
- rules for comparing, 32
- as values, 57
- variable arguments becoming, 44
- zip function producing, 21–23

TypeError
- exceptions, for keyword-only arguments, 53–54
- rejecting iterators, 41–42

tzinfo class, for time zone operations, 164–165

U

unicode instances, for character sequences, 5–7
Unit tests, 207
unittest built-in module, for writing tests, 205–207
UNIX timestamp, in time conversions, 163–165
Unordered dictionaries, 167
up command, of interactive debugger, 209
UTC (Coordinated Universal Time), in time conversions, 162–165
Utility classes, mix-in, creating hierarchies of, 77–78

V

Validation code, metaclasses running, 105–108
ValueError exceptions, 30–31, 184
Values
- from iterators, 40–42
- tuples as, 57
- validating assignments to, 89

Variable positional arguments
- keyword arguments and, 47–48
- reducing visual noise, 43–45

Variable scopes, closure interaction with, 31–36
--version flag, determining version of Python, 1–2
Virtual environments
- pyvenv tool creating, 194–196
- reproducing, 196–197

virtualenv command-line tool, 194
Visual noise, positional arguments reducing, 43–45

W

WeakKeyDictionary, purpoose of, 99
weakref module, building descriptors, 113
while loops, else blocks following, 23–25
Whitespace, importance of, 3
Wildcard imports, 183
with statements
- mutual-exclusion locks with, 153–154
- for reusable try/finally blocks, 154–155
- as target values and, 155–156

wraps helper function, from functools, for defining decorators, 152–153

Y

yield expression
  in coroutines, 137–138
  in generator functions, 37
  use in contextlib, 155
yield from expression, unsupported in Python 2, 144

Z

ZeroDivisionError exceptions, 30–31, 51
zip built-in function
  for iterators of different lengths, 170
  processing iterators in parallel, 21–23
zip_longest function, for iterators of different length, 22–23, 170